NORTH-HOLLAND

# ON DECOMPOSITIONS OF CHAIN DATALOG PROGRAMS INTO $\mathcal{P}$ (LEFT-)LINEAR 1-RULE COMPONENTS

GUOZHU DONG AND SEYMOUR GINSBURG

▷ As an approach to optimization, this paper examines the decomposition of chain Datalog programs into $\mathcal{P}$ (left-)linear sequences of 1-rule programs. The notion of $\mathcal{P}$ (left-)linear, introduced here, encompasses numerous special (left-) linear forms and includes the traditional (left) linear as a subcase. The decompositions are first characterized in terms of properties of associated context-free languages. More specific characterizations are provided for three types of $\mathcal{P}$ (left-)linear decompositions with 1-rule components, and the corresponding decision problems considered. Finally, arbitrarily large, inherently nondecomposable, $\mathcal{P}$-linear size-prime programs are exhibited. ◁

## 1. INTRODUCTION

It is widely agreed that Datalog, i.e., logic programs without negation and function symbols, is an elegant deductive query language for databases. The efficiency issue for evaluating Datalog queries has attracted a great deal of attention in the database community for the past few years. One particular approach dealing with this issue

Guozhu Dong gratefully acknowledges the support of the Australian Research Council. Seymour Ginsburg was supported in part by the National Science Foundation under Grant IRI-8920930; a portion of the work by this author was done while visiting the University of Melbourne through the support of an ARC grant to the first author.

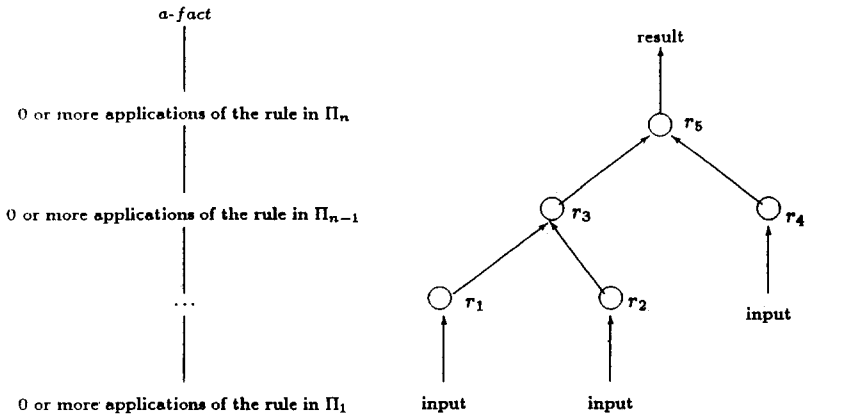
*Address correspondence to* Guozhu Dong, Computer Science Department, University of Melbourne, Parkville, Vic. 3052, Australia, dong@cs.mu.oz.au; Seymour Ginsburg, Computer Science Department, University of Southern California, Los Angeles, CA 90089-0781, USA, ginsburg@pollux.usc.edu.

Received October 1992; accepted October 1994.

*THE JOURNAL OF LOGIC PROGRAMMING*


655 Avenue of the Americas, New York, NY 10010

0743-1066/95/$9.50
SSDI 0743-1066(94)00027-4




Rule applications are layered in derivation trees.

Sequential processing.

(a)

Each processor responsible for one rule. Facts flow from children to parent only.

Parallel processing.

(b)

**FIGURE 1.** Advantages of decomposition in sequential and parallel processing.

decomposes queries into sequences of simpler and smaller clusters [4, 7]. Most appealing are decompositions with 1-rule component programs. From a sequential processing point of view, each rule $r$ in such a decomposition is evaluated after those on which $r$ depends (see Figure 1(a)), thus removing unwanted "spaghetti" interactions among rules. From a parallel processing point of view, such a decomposition allows a tree architecture (based on rule dependence) of parallel processors, where (i) each processor is responsible for exactly one rule, and (ii) facts are passed only from children processors to parent processors (see Figure 1(b)). The advantages of decompositions with 1-rule component programs have also been recognized in two other recent studies [22, 25]. The purpose of the current paper is to investigate the transformation of chain Datalog programs into several desirable forms of decompositions with 1-rule components.

Chain Datalog programs were first defined in [29], and have received much attention recently. Indeed, such programs were examined in [2, 1, 3], and were used as examples in many papers [18–20, 22, 23, 27]. This extensive attention can perhaps be attributed in part to the existence of some interesting connections between chain Datalog programs and context-free languages [2, 3, 24, 26, 29].

The theme of the paper is "$\mathcal{P}$ (left-)linear decompositions." It depends on two major notions, namely, "decompositions" and "$\mathcal{P}$ (left-)linear." The decomposition approach in general has been considered by several authors lately. The decomposition of general Datalog programs was examined in [4, 7] and the decomposition of databases in [5]. In [22], it was claimed that decomposability based on commutativity can reduce the complexity of "counting" methods for query evaluation from exponential to polynomial. The decidability and complexity of decomposition into sequences of rules were examined under the name "rule sequencability" [24, 25]. There have also been investigations on factorization of argument positions in predicates [20] and on factorization of rules (into common generating rules with smaller bodies) [21].

(Left-)linear Datalog programs have attracted much attention since they allow special efficient evaluations [1, 6, 15, 22, 23, 28, 30]. However, the customary concept of linearity treats IDB predicates uniformly. In practice, one may want to emphasize linearity in predicates with special properties, and ignore multiplicity in other predicates. The notion of $\mathcal{P}$ (left-)linear allows one to capture such situations, and encompasses numerous interesting subcases (including the "traditional" linear).

The technical contributions center around three themes: characterizations, decidability, and primality (or nondecomposibility). Several results characterize decomposability in terms of properties of the associated context-free languages. More specific characterizations are provided for three types of $\mathcal{P}$-linear decompositions with 1-rule component programs. The decision problems for these decompositions are considered. Arbitrarily large (inherently nondecomposable) linear "size-prime" programs are exhibited. For all the results except Theorems 3.3 and 3.4 (which concern the existence of some hierarchy of $\mathcal{P}$-linear programs), each theorem about $\mathcal{P}$-linear has a counterpart for $\mathcal{P}$ left-linear. The fact that there are no counterparts for Theorems 3.3 and 3.4 can be interpreted as follows: $\mathcal{P}$-linear is powerful enough to yield complicated interactions among an arbitrarily large number of rules with a common head, for arbitrary $\mathcal{P}$; but $\mathcal{P}$ left-linear guarantees simple interactions among rules with a common head, for some $\mathcal{P}$ at least. In addition, there are some results concerning $\mathcal{P}$ left-linear, specifically, Theorems 4.2 and 4.4, which do not have counterparts for $\mathcal{P}$-linear. Finally, all the results hold if "left" is replaced by "right."

Obviously, this investigation employs the vehicle of context-free languages. However, the fundamental tool used, namely, 1-sequential, is new. Furthermore, the concept of decomposition for Datalog programs was only recently introduced and studied, and the concept of $\mathcal{P}$ (left-)linear is new. The results of this paper are useful for both Datalog query optimization and for the theory of context-free languages. They are *not* interpretations of previously known results. Furthermore, the proofs in this paper present new techniques for Datalog optimization. Indeed, complicated examples such as Example 4.1 show the systematic construction of decompositions using the new techniques. Without these new techniques, it is difficult to perceive the existence of such decompositions, let alone how to construct them.

Organizationally, the paper consists of four sections (plus this Introduction) and an Appendix. Section 2 gives the preliminaries. Section 3 introduces the notion of $\mathcal{P}$ (left-)linear, and investigates general $\mathcal{P}$ (left-)linear decompositions with 1-rule components. Section 4 studies three special $\mathcal{P}$ (left-)linear decompositions with 1-rule components. Concluding remarks are made in Section 5.

## 2. PRELIMINARIES

This section first reviews the fundamentals of chain Datalog programs. It then presents the well-known correspondence between such programs and context-free languages. Finally, it introduces a major notion of the paper, namely, decomposition of programs.

### *2.1. Chain Datalog Programs*

We now define "chain Datalog programs" and "IDB" and "EDB" predicates.

We assume the existence of three pairwise disjoint infinite sets of *constants*, *variables*, and *predicates*. Constants are denoted by $x$, $y$, and $z$ (possibly with subscripts and superscripts[1]), variables by $X$, $Y$, and $Z$, and predicates by $p$, $q$, $a$, $b$, $c$, and $d$. A *term* is either a variable or a constant. An *atom* is a formula of the form $q(t_1, \ldots, t_k)$, where $q$ is a predicate and $t_1, \ldots, t_k$ are terms.

Although some of our discussion applies to general Datalog programs, we shall restrict our attention exclusively to "chain" Datalog programs.

*Definition 2.1.* A *chain rule* (or simply *rule*) is an expression of the form

$$q(X, Z) :- q_1(X, Y_1), q_2(Y_1, Y_2), \ldots, q_{k+1}(Y_k, Z).$$

where $k \geq 0$, $q$ and each $q_i$ are predicates, and $X$, $Z$, and each $Y_i$ are distinct variables. Here $q(X, Z)$ is the *head* and $q_1(X, Y_1), q_2(Y_1, Y_2), \ldots, q_{k+1}(Y_k, Z)$ the *body*. (The body becomes $q_1(X, Z)$. when $k = 0$.) A *chain Datalog program*, or simply *program*, is a finite (possibly empty) set of rules. Rules are denoted by $r$ and programs by $\Pi$.

Note that each chain rule (i) contains no constants and (ii) has at least one atom in its body. Also observe that all chain rules are "safe" [27]. Finally, because our interest is in chain Datalog programs, all atoms are considered to be of the form $q(t_1, t_2)$.

The predicates occurring in a program $\Pi$ are divided into two categories:

- IDB (Intentional Database) predicates are those that appear in rule heads, and possibly in rule bodies; $p$, $p_1$, $p_2$, ... usually denote IDB predicates.
- EDB (Extensional Database) predicates are those that appear in rule bodies only; $a$, $b$, $c$, and $d$ (possibly with subscripts) always denote EDB predicates. The symbols $q, q_1, q_2, \ldots$ denote either IDB or EDB predicates. The set of IDB (EDB resp.) predicates of $\Pi$ is denoted by $IDB(\Pi)$ ($EDB(\Pi)$ resp.).

To illustrate, consider the following program [19]:

$$buys(X, Z) :- friend(X, Y), buys(Y, Z).$$
$$buys(X, Z) :- idol(X, Y), buys(Y, Z).$$
$$buys(X, Z) :- perfect\ for(X, Y).$$

Here, *buys* is the only IDB predicate, and *friend*, *idol*, and *perfect for* are EDB predicates.

Due to the notion of "decomposition" defined later, a predicate $p_i$ may be an IDB predicate in one program and an EDB predicate in another program.

### 2.2. Semantics of Datalog Programs

We now define the semantics of programs and discuss the class of queries of concern to us.

A *fact* is an atom both of whose terms are constants. Given a program $\Pi$, an *interpretation* is a finite set of facts over some of the predicates in $IDB(\Pi) \cup$

[1]Subscripts and superscripts also apply to the symbols used for variables, predicates, etc.

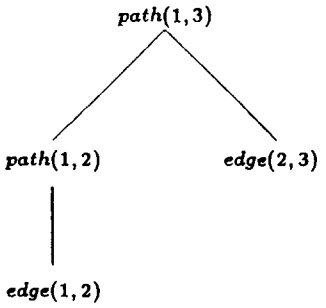



**FIGURE 2.** A derivation tree.

$EDB(\Pi)$, an *EDB fact* is a fact over an EDB predicate, and an *EDB instance* is a finite set of facts over some EDB predicates.

A *substitution* is a mapping $\theta$ from terms to terms which is the identity on all constants. For each substitution $\theta$ and atom $q(t_1, t_2)$, let $\theta(q(t_1, t_2))$ denote the atom $q(\theta(t_1), \theta(t_2))$. A substitution $\theta$ is called a *unifier* for two atoms $A$ and $B$ if $\theta(A) = \theta(B)$. Two atoms are *unifiable* if there is a unifier for them.

The computation of programs can be represented using the so-called "derivation trees." Given a program $\Pi$, a $\Pi$-*derivation tree* (or *derivation tree* if $\Pi$ is understood) for a fact $f$ is an ordered rooted tree, with atoms as nodes and with edges between parents and children, such that

- $f$ is the root.
- For every nonleaf node $A'_1$, if its children are $B'_1, \ldots, B'_k$, then there is a rule (called the rule *associated with* $A'_1$) $A_1 :- B_1, \ldots, B_k$ in $\Pi$ and substitution $\theta$ such that (i) $A'_1 = \theta(A_1)$, and (ii) $B'_i$ is a fact and is equal to $\theta(B_i)$ for each $i$.

To each program $\Pi$, there corresponds a function $T^\omega_\Pi$ from the set of interpretations (of $\Pi$) to itself. Specifically, for each interpretation $I$, let

$$T^\omega_\Pi(I) = I \cup \{A \mid \text{there is a derivation tree for } A \text{ whose leaves are in } I\}.$$

(Note that the set to the right of the equal sign is finite.)

To illustrate, let $\Pi$ be the following program:

$$path(X, Z) :- edge(X, Z).$$
$$path(X, Z) :- path(X, Y), edge(Y, Z).$$

For $I = \{edge(1,2), edge(2,3), edge(3,4)\}$, we have $T^\omega_\Pi(I) = \{edge(1,2), edge(2,3), edge(3,4), path(1,2), path(2,3), path(3,4), path(1,3), path(2,4), path(1,4)\}$. A derivation tree for $path(1,3)$ is given in Figure 2.

The set $T^\omega_\Pi(I)$ is also called the "success set" in the literature, and (by [17, Theorem 8.3]) is the minimum "Herbrand model" of $\Pi \cup I$ (i.e., is a subset of every Herbrand model of $\Pi \cup I$). We differ slightly from the traditional approach, where $I$ must be an EDB instance and is considered as part of the program. Nevertheless, $I$ is part of the output in both treatments. The difference is needed for the consideration of computing with "decompositions."

Given a program $\Pi$, we shall be interested in queries of the form $p(X, Y)$, where $X$ and $Y$ are distinct variables and $p$ is an arbitrary predicate. For each interpretation $I$ of $\Pi$, the *answer* for $\Pi$ on $I$ with respect to $p$ is the set $\{p(x, y) \mid p(x, y)\ in\ T^\omega_\Pi(I)\}$.

For example, let $\Pi$ be the following program:

$$\begin{aligned} p(X,Z) &:- p'(X,Y_1), p(Y_1,Y_2), d(Y_2,Z). \\ p'(X,Z) &:- a(X,Z). \\ p'(X,Z) &:- b(X,Z). \\ p(X,Z) &:- c(X,Z). \end{aligned}$$

For predicate $p$ and EDB instance $I = \{a(1,2), c(2,3), d(3,4)\}$, the answer set is $\{p(2,3), p(1,4)\}$. (Note that $T_\Pi^\omega(I) = I \cup \{p'(1,2), p(2,3), p(1,4)\}$.)

We now define the "equivalence" of programs. In order to be able to compare programs with different EDB predicates, it is necessary to allow the interpretation of a program to have facts on some "irrelevant" predicates. For example, suppose we have a program $\Pi$ and an "optimized" version $\Pi'$ of $\Pi$. In $\Pi$, there may be an EDB predicate $a$ which is "irrelevant" to the situation in the sense that facts over $a$ do not contribute to the query answer. The optimized version $\Pi'$ may discard this irrelevant EDB predicate $a$. Thus, for each program $\Pi$, we extend $T_\Pi^\omega$ so that it applies to each set $I_1 \cup I_2$ of facts, where (i) $I_1$ is an EDB instance of $\Pi$, and (ii) $I_2$ is a set of facts over a set of predicates disjoint from $IDB(\Pi)$. Here, $I_2$ is the set of facts for the irrelevant predicates. Note that the answer for $\Pi$ on $I_1 \cup I_2$ only depends on $I_1$.

*Definition 2.2.* For each predicate $p$, two programs $\Pi_1$ and $\Pi_2$ are called *$p$-equivalent* if (i) $EDB(\Pi_1) \cup EDB(\Pi_2)$ is disjoint from $IDB(\Pi_1) \cup IDB(\Pi_2)$, and (ii) for each EDB instance $I$ of $\Pi_1 \cup \Pi_2$, $\{p(x,y) \mid p(x,y)$ *in* $T_{\Pi_1}^\omega(I)\} = \{p(x,y) \mid p(x,y)$ in $T_{\Pi_2}^\omega(I)\}$.

### *2.3. A CFL Representation of Chain Datalog Programs*

We shall use a close correspondence between chain Datalog programs and context-free grammars. This relationship has been used in the literature, for example, [3, 29], to analyze chain Datalog programs.

We first recall some basic notions of context-free grammars and context-free languages, modified slightly for our purpose. A *context-free grammar*, or *grammar* for short, is a 4-tuple $G = (V, \Sigma, P, p_1)$, where $V$ and $\Sigma$ are disjoint finite sets of symbols, $p_1$ is in $V$, and $P$ is a finite (possibly empty) set of expressions of the form $p \to u$, with $p$ in $V$ and $u \in (V \cup \Sigma)^*$. Elements of $\Sigma$ and $V$ are called *terminals* and *variables*, respectively, $p_1$ is called the *start* variable, and $p \to u$ is called a *production* with $p$ as *head* and $u$ as *body*. Let $\Rightarrow_G$ be the relation on $(V \cup \Sigma)^*$ defined by $y \Rightarrow_G z$ if there exist $u, v, p, x$ such that $y = upv$, $z = uxv$, and $p \to x$ is in $P$. Let $\Rightarrow_G^*$ be the reflexive transitive closure of $\Rightarrow_G$. The set $L(G) = \{w$ in $\Sigma^* \mid p_1 \Rightarrow_G^* w\}$ is called the *language generated* by $G$. A set $L$ is said to be a *context-free language* or simply a *language*, if $L = L(G)$ for some context-free grammar $G$. Since context-free languages are used here strictly as a tool for studying chain Datalog programs, we shall almost exclusively discuss the $\Lambda$-free grammars and languages, where $\Lambda$ denotes the empty word. (By a $\Lambda$-free grammar is meant a grammar with no productions of the form $p' \to \Lambda$.) Hence, unless stated otherwise, by a grammar or language is meant a $\Lambda$-free grammar or language. Without loss of generality, we assume that (i) there is no production of the form $p \to p$, and (ii) if $L(G) \neq \emptyset$, then for each variable $p$, there is $u \in \Sigma^+$ such

that $p \Rightarrow_G^* u$. A context-free grammar which may have $\Lambda$-rules will be called an *extended* (context-free) grammar and the associated language an *extended* language.

By (i) ignoring Datalog variables, and (ii) treating IDB predicates as grammar "variables" and EDB predicates as grammar "terminals," chain rules correspond to context-free productions and derivations of chain Datalog programs correspond to derivations of context-free grammars. For example, the chain rule

$$p(X, Z) :- a(X, Y_1), p(Y_1, Y_2), b(Y_2, Z).$$

corresponds to $p \rightarrow apb$. If we make a new copy by subscripting variables with 1, then unify its head with $p(Y_1, Y_2)$ and "substitute," we obtain an expanded chain rule

$$p(X, Z) :- a(X, Y_1), a(Y_1, Y_{11}), p(Y_{11}, Y_{21}), b(Y_{21}, Y_2), b(Y_2, Z).$$

The above step obviously corresponds to the derivation $p \Rightarrow^* aapbb$.

*Notation.* For each program $\Pi$ and predicate $p_1$, let $G_{\Pi,p_1}$ be the context-free grammar $(V, \Sigma, P, p_1)$, where (i) $V = IDB(\Pi)$, (ii) $\Sigma = EDB(\Pi)$, and (iii) $P = \{p \rightarrow q_1 \cdots q_n \mid \Pi$ contains a rule of the form $p(X, Z) :- q_1(X, Y_1), q_2(Y_1, Y_2), \ldots, q_n(Y_{n-1}, Z).\}$.

For example, let $\Pi$ consist of the following rules:

$$p_1(X, Z) :- a(X, Y_1), p_1(Y_1, Y_2), d(Y_2, Z).$$
$$p_1(X, Z) :- b(X, Y_1), p_1(Y_1, Y_2), d(Y_2, Z).$$
$$p_1(X, Z) :- c(X, Z).$$

Then $G_{\Pi,p_1} = (\{p_1\}, \{a, b, c, d\}, \{p_1 \rightarrow ap_1d, p_1 \rightarrow bp_1d, p_1 \rightarrow c\}, p_1)$.

### *2.4. Decomposition*

As mentioned in the Introduction, the theme of the paper is "$\mathcal{P}$ (left-)linear decompositions," and depends on two major notions. We now define the first of the two major notions, namely, "decomposition."

*Definition 2.3.* For each predicate $p$, a sequence $\Pi_1 \cdots \Pi_n$ $(n \geq 1)$ of programs is called a *p-decomposition* of a program $\Pi$ if $\{p(x, y)' \mid p(x, y)$ in $T^\omega_{\Pi_1} \circ \cdots \circ T^\omega_{\Pi_n}(I)\} = \{p(x, y)' \mid p(x, y)$ in $T^\omega_\Pi(I)\}$ for each EDB instance $I$ of $\Pi$. (Here, $\circ$ denotes mapping composition, and the component mappings are applied from right to left.) Each $\Pi_i$ is called a *component program* or simply *component* of the $p$-decomposition.

Note that every program is a degenerate decomposition of length one.

We first present a simple example. (More involved examples appear in the text.) Let $\Pi$ be the following:

$$r_1: p(X, Z) :- a(X, Y_1), a(Y_1, Y_2), p(Y_2, Y_3), b(Y_3, Y_4), b(Y_4, Y_5), b(Y_5, Z).$$
$$r_2: p(X, Z) :- a(X, Y_1), a(Y_1, Y_2), a(Y_2, Y_3), p(Y_3, Y_4), b(Y_4, Y_5), b(Y_5, Z).$$
$$r_3: p(X, Z) :- c(X, Z).$$

Then $\{r_1\}\{r_2\}\{r_3\}$ is a $p$-decomposition with 1-rule components[2] for $\Pi$. (This is verified after Theorem 4.6.) For each interpretation $I$, the answer for $\Pi$ on $I$ with respect to $p$ is determined by first computing the fixpoint $F3$ of $r_3$ on $I$, followed by the fixpoint $F2$ of $r_2$ on $F3$, and finally the fixpoint of $r_1$ on $F2$. There is no need to consider computations where $r_1$ is applied first, followed by the application of other rules. (Observe that $r_1$ and $r_2$ commute, and $L(G_{\Pi,p}) = \{a^{3m+2n}b^{2m+3n} \mid m, n \geq 0\}$.)

We now compare decompositions studied in [7], here called "uniform decompositions," with $p$-decompositions. A sequence $\Pi_1 \cdots \Pi_n$ of programs is called a *uniform decomposition* of a program $\Pi$ if $T^\omega_{\Pi_1} \circ \cdots \circ T^\omega_{\Pi_n}(I) = T^\omega_\Pi(I)$ for each interpretation $I$ of $\Pi$. The similarity between the two kinds of decompositions is the ordered, compositional manner of computation of the component programs. The differences are: (i) uniform decompositions take as input interpretations of both IDB and EDB predicates of $\Pi$, whereas $p$-decompositions take as input only EDB instances of $\Pi$; (ii) uniform decompositions "simulate" $\Pi$ for every IDB predicate of $\Pi$, whereas $p$-decompositions "simulate" $\Pi$ only for $p$; and (iii) uniform decompositions do not use "scratch-paper" predicates (i.e., predicates not in $\Pi$), whereas $p$-decompositions may use such predicates (as is done in most other papers).

The purpose of decomposition is to divide programs into smaller clusters. By doing so, some unwanted interactions among rules may be removed. Thus, decomposition helps to achieve more efficient evaluations of programs as queries on databases. The separation of interactions may also help a user to better understand the programs.

Most appealing are decompositions with 1-rule components. (Decompositions with 1-rule components in the uniform sense were studied in [4, 7].) Indeed, the evaluation of a sequence $\Pi_1 \cdots \Pi_n$ of 1-rule programs can be accomplished in a completely layered manner, as illustrated in Figure 1(a). The purpose of this paper is to investigate such decompositions, especially with respect to situations where the component 1-rule programs have some particularly desirable forms.

## 3. GENERAL $\mathcal{P}$ (LEFT-)LINEAR DECOMPOSITIONS

This section introduces the theme of our investigation, namely, "$\mathcal{P}$ (left-)linear decompositions." It then studies *general* $\mathcal{P}$ (left-)linear decompositions with 1-rule components. (Section 4 will explore *specialized* $\mathcal{P}$ (left-)linear decompositions with 1-rule components.) Our major result of the section (Theorem 3.1) characterizes the existence of such decompositions by a so-called "1-sequentiality" property of context-free languages. The problem of whether an arbitrary program has a $\mathcal{P}$ (left-)linear decomposition with 1-rule components is then shown to be undecidable for a degenerate $\mathcal{P}$. For each $\mathcal{P}$, (i) arbitrarily large "$\mathcal{P}$-linear size-prime" programs are exhibited, and (ii) $\mathcal{P}$-linear decompositions with 1-rule components are proven to be strictly less general than $\mathcal{P}$-linear decompositions with 2-rule "uni-head" components. It will be shown in Section 4 (Corollary 4.2) that (i) and (ii) are false for some $\mathcal{P}$ if "linear" is replaced by "left linear."

We begin with the second of our two major notions of the paper, namely, $\mathcal{P}$ (left-)linear. (Left-)linear programs have attracted much attention since they allow

[2]For each integer $m$, a program is called an *$m$-rule* program if it contains at most $m$ rules.

special efficient evaluations. However, the customary notion of (left) linearity treats IDB predicates uniformly. In practice, one may want to emphasize (left) linearity in IDB predicates with special properties, and ignore multiplicity in other predicates. The notion of $\mathcal{P}$ (left-)linear allows one to capture such situations.

A *variable-renaming mapping* is a function which is one-to-one from variables to variables and is the identity on constants and predicates. Each variable-renaming mapping $f$ is extended homomorphically to rules by letting $f$ be the identity on the "punctuation" symbols. A rule $r'$ is a *variable renaming* of a rule $r$ if there is a variable-renaming mapping $f$ such that $r' = f(r)$. A program $\Pi'$ is a *variable renaming* of a program $\Pi$ if each $r'$ in $\Pi'$ is a variable renaming of some $r$ in $\Pi$.

Note that the statement "$\Pi'$ is a variable renaming of $\Pi$" is obviously implied by "there exists a variable-renaming mapping $f$ such that each $r'$ in $\Pi'$ is equal to $f(r)$ for some $r$ in $\Pi$." But the converse is false:

$$\Pi' = \{p(X,Z) :- a(X,Z)., p(X,Y) :- a(X,Y).\}$$

is a variable renaming of $\Pi = \{p(X,Y) :- a(X,Y).\}$, but there is no variable renaming mapping $f$ such that each $r'$ in $\Pi'$ is equal to $f(r)$ for some $r$ in $\Pi$.

If $r'$ is a variable renaming of $r$, then $r$ is a variable renaming of $r'$. However, it is not always true that if a program $\Pi'$ is a variable renaming of a program $\Pi$, then $\Pi$ is a variable renaming of $\Pi'$. For example, $\{p(X,Y) :- a(X,Y).\}$ is a variable renaming of $\{p(X,Z) :- a(X,Z)., p_1(X,Y) :- b(X,Y).\}$, but not vice versa.

*Definition 3.1.* A *limiting mapping* is a function $\mathcal{P}$ from the Cartesian product of the family of programs and the family of rules to the family of finite sets of predicates satisfying the following: For all programs $\Pi$ and $\Pi'$ and all rules $r$ and $r'$, where $\Pi$ and $\Pi'$ are variable renamings of each other and $r'$ is a variable renaming of $r$, (i) $\mathcal{P}(\Pi, r) \subseteq IDB(\Pi)$ and (ii) $\mathcal{P}(\Pi, r') = \mathcal{P}(\Pi', r)$.

Intuitively, condition (i) says that $\Pi$ in $\mathcal{P}(\Pi, r)$ provides the predicates of interest when $r$ is considered. Condition (ii) says that $\mathcal{P}$ depends only on the predicate patterns in rules and programs, and not on the particular variables used. This condition will ensure a direct correspondence between programs and grammars.

For example, three limiting mappings are $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$ defined (for each program $\Pi$ and rule $r$) by $\mathcal{P}_1(\Pi, r) = IDB(\Pi)$, $\mathcal{P}_2(\Pi, r) = IDB(\{r\})$ if $IDB(\{r\}) \subseteq IDB(\Pi)$ and $\mathcal{P}_2(\Pi, r) = \emptyset$ otherwise, and $\mathcal{P}_3(\Pi, r) = \emptyset$. More examples will be given later.

Limiting mappings are used to specify the emphasized predicates (and are hence denoted by $\mathcal{P}$). To formally define this, for each limiting mapping $\mathcal{P}$ and program $\Pi$, let $\mathcal{P}_\Pi$ be the mapping defined by $\mathcal{P}_\Pi(r) = \mathcal{P}(\Pi, r)$ for each rule $r$.

*Definition 3.2.* For each limiting mapping $\mathcal{P}$ and program $\Pi'$, a program $\Pi$ is called *$\mathcal{P}_{\Pi'}$-linear* if, for each $r \in \Pi$, at most one predicate in $\mathcal{P}_{\Pi'}(r)$ occurs in the body of $r$, and it can occur there at most once; and $\Pi$ is called *$\mathcal{P}_{\Pi'}$ leftmost-linear*, abbreviated *$\mathcal{P}_{\Pi'}$ left-linear*, if $\Pi$ is $\mathcal{P}_{\Pi'}$-linear and, for each $r \in \Pi$, only the leftmost predicate in the body of $r$ can appear in $\mathcal{P}_{\Pi'}(r)$.

The phrase "left" indicates the location where the emphasized predicates may occur. Almost all results in this paper apply to both the linear and left linear cases.

Every result on left linear has an obvious counterpart by changing the phrase "left" to the phrase "right."

The notion of $\mathcal{P}$-linear encompasses many different forms of linear which can be examined and utilized fruitfully. For example, suppose $\Pi$ is a program. For $\mathcal{P}$ defined by $\mathcal{P}(\Pi', r) = IDB(\Pi')$ for all $\Pi'$ and $r$, $\Pi$ is $\mathcal{P}_\Pi$ (left-)linear iff $\Pi$ is (left) linear in the traditional sense (which we shall simply call *(left) linear*). For $\mathcal{P}$ defined by $\mathcal{P}(\Pi', r) = \emptyset$ for each $\Pi'$ and $r$, $\Pi$ is *always* $\mathcal{P}_{\Pi'}$ (left)-linear for all $\Pi'$. Three other types of limiting mappings will appear later in the paper, and the associated linearities will be studied fruitfully in some depth.

Using the major concepts of decomposition and $\mathcal{P}$ (left-)linear, we now present the central notion of this paper.

*Definition 3.3.* For each limiting mapping $\mathcal{P}$ and predicate $p$, a $p$-decomposition $\Pi_1 \cdots \Pi_n$ is called $\mathcal{P}$ *(left-)linear* if each $\Pi_i$ is $\mathcal{P}_{\cup_{j=1}^n \Pi_j}$ (left-)linear.

To establish the main result, we need several concepts and three technical lemmas.

The notions of "depend on" and "strongly connected clusters" of rules abstract some syntactic relationships.

*Definition 3.4.* For all programs $\Pi$ and rules $r_1$ and $r_2$ in $\Pi$, $r_1$ is said to *depend on* $r_2$ (in $\Pi$), denoted $r_1 \succ_\Pi r_2$, if either the predicate occurring in the head of $r_2$ occurs in the body of $r_1$, or there is a rule $r$ in $\Pi$ such that $r_1 \succ_\Pi r$ and $r \succ_\Pi r_2$. A *strongly connected cluster*, or SCC for short, of $\Pi$ is a maximal subset $\Pi'$ of $\Pi$ such that $r \succ_\Pi r'$ for all distinct rules $r$ and $r'$ in $\Pi'$.

Clearly, the SCCs of a program form a partition of that program.

By [7, Theorem 2.4], each program $\Pi$ has a uniform decomposition into SCCs of $\Pi$. Obviously, each uniform decomposition is a $p$-decomposition for every predicate $p$. Thus, we have our first technical lemma.

*Lemma 3.1. For each program $\Pi$ and predicate $p$, $\Pi$ is $p$-decomposable into programs which are SCCs of $\Pi$.*

Note that Lemma 3.1 does not say anything about dividing SCCs. Needless to say, the decomposition of SCCs into smaller pieces is of interest and useful. Several such results were obtained in [7], and more will be provided in this paper.

To present our second technical lemma, we need four concepts. The first two are concerned with structural aspects of decompositions. Specifically, a triple $(q, i, j)$ is called a *reverse dependency* in $\Pi_1 \cdots \Pi_n$ if $1 \leq i < j \leq n$, and $q$ occurs in the head of a rule in $\Pi_i$ and in the body of a rule in $\Pi_j$. A sequence $\Pi_1 \cdots \Pi_n$ is *normal* if each $\Pi_i$ is an SCC of itself, i.e., is an SCC of $\Pi_i$, and there is no reverse dependency in $\Pi_1 \cdots \Pi_n$.

Suppose $\Pi_1 \cdots \Pi_n$ is a normal sequence. Then each $\Pi_i$ is obviously an SCC of $\cup_{j=1}^n \Pi_j$, so that $\{\Pi_1, \ldots, \Pi_n\}$ is a partition of $\cup_{j=1}^n \Pi_j$.

The other two concepts are related to the relabeling of predicates in programs. A *predicate-renaming mapping* is a function on the predicates, variables, and constants which is one-to-one from predicates to predicates and is the identity on constants and variables. (Thus, it only changes predicates.) Each predicate-renaming mapping is extended homomorphically to rules by letting $f$ be the identity on the

"punctuation" symbols. A program $\Pi'$ is a *predicate renaming* of a program $\Pi$ if there is a predicate-renaming mapping $f$ such that $\Pi' = \{f(r) \mid r \in \Pi\}$.

For example, $\Pi'$ is a predicate renaming of $\Pi$, where $\Pi$ is

$$p_1(X,Z) :- p_1(X,Y), p_3(Y,Z).$$
$$p_1(X,Z) :- p_3(X,Z).$$

and $\Pi'$ is

$$p_1(X,Z) :- p_1(X,Y), p_2(Y,Z).$$
$$p_1(X,Z) :- p_2(X,Z).$$

It is easy to see that predicate renamings are closed under composition.

The second technical lemma ensures the existence of some normal decomposition if there is a decomposition. This result will play an essential role in relating decomposability and "sequentiality". (This lemma also applies to nonchain programs.)

*Lemma 3.2. Let $\Pi$ be a program and $p$ a predicate. Suppose there is a $p$-decomposition $\Pi_1 \cdots \Pi_n$ of $\Pi$. Then there is a normal $p$-decomposition $\Pi_1^* \cdots \Pi_k^*$ of $\Pi$ where, for $1 \le j \le k$, either (a) $\Pi_j^*$ has the form $\{q_1(X,Y) :- q_2(X,Y).\}$; or (b) there is an $i$ $(1 \le i \le n)$ such that $\Pi_j^*$ is a predicate renaming of $\Pi_i$.*

PROOF. By Lemma 3.1, we may assume that each $\Pi_i$ is an SCC of itself. For each sequence $\Pi_1'' \cdots \Pi_k''$, let $RD(\Pi_1'' \cdots \Pi_k'')$ be the following (possibly empty) set:

$$\{(q,i,j) \mid (q,i,j) \text{ is a reverse dependency in } \Pi_1'' \cdots \Pi_k''\}.$$

Suppose $RD(\Pi_1 \cdots \Pi_n) \neq \emptyset$. We now transform the $p$-decomposition $\Pi_1 \cdots \Pi_n$ into another $p$-decomposition which is "smaller" as measured by the $RD$ function. (See Claims A and B below.) Indeed, let $(q,i',j')$ be a lexical minimum in $RD(\Pi_1 \cdots \Pi_n)$, that is, for all $(q^*,i,j)$ in $RD(\Pi_1 \cdots \Pi_n)$, either $i' = i$ and $j' \le j$ or $i' < i$. Since $RD(\Pi_1 \cdots \Pi_n) \neq \emptyset$, $(q,i',j')$ exists. Let $q'$ be a new predicate, and $\Pi_{new} = \{q(X,Y) :- q'(X,Y).\}$. For each $j \ge j'$, let $\Pi_j'$ be obtained from $\Pi_j$ by renaming $q$ with $q'$, i.e., by replacing each occurrence of $q$ with $q'$.

*Claim A.* $\Pi_1 \cdots \Pi_{j'-1} \Pi_{new} \Pi_{j'}' \cdots \Pi_n'$ is a $p$-decomposition of $\Pi$.

PROOF. We first need to introduce the following concept. Given a sequence $\Pi_1 \cdots \Pi_n$, a $(\cup_{i=1}^n \Pi_i)$-derivation tree is called a $\Pi_1 \cdots \Pi_n$-*derivation tree* if there is a function $N$ from the internal nodes to $\{1,\ldots,n\}$ satisfying the following condition: For each node $A$ and each son $B$ of $A$, (i) $\Pi_{N(A)}$ contains the rule associated with $A$, and (ii) $N(B) \ge N(A)$ if $B$ is also an internal node.

Turning to the proof of Claim A, suppose we are given a $\Pi_1 \cdots \Pi_n$-derivation tree for a $p$ fact. A $\Pi_1 \cdots \Pi_{j'-1} \Pi_{new} \Pi_{j'}' \cdots \Pi_n'$-derivation tree for the same $p$ fact is obtained as follows: (i) For each node $A$ associated with a rule in $\Pi_j$ $(j' \le j \le n)$, rename all the occurrences of $q$ with $q'$. (ii) For each node $B$ associated with a rule in some $\Pi_i$ $(1 \le i < j')$ and each child $C$ of $B$ where $q'$ occurs, insert between $B$ and $C$ a node $C'$ obtained by renaming $q'$ in $C$ with $q$. (iii) If the result of (i) and (ii) is a derivation tree with a root node $A$ of the form $q'(x,y)$, add $q(x,y)$ as the parent

of $A$. (For (iii), the rule in $\Pi_{new}$ is used.) Similarly, a $\Pi_1 \cdots \Pi_{j'-1}\Pi_{new}\Pi'_{j'} \cdots \Pi'_n$-derivation tree can be transformed into a $\Pi_1 \cdots \Pi_n$-derivation tree by renaming $q'$ with $q$ and removing the translation step from $q'$ to $q$ using the rule in $\Pi_{new}$.

Claim B. $|RD(\Pi_1 \cdots \Pi_{j'-1}\Pi_{new}\Pi'_{j'} \cdots \Pi'_n)| < |RD(\Pi_1 \cdots \Pi_n)|$.[3]

PROOF. Let $h$ be the mapping defined, for each triple $(q^*, i^*, j^*)$ in $RD(\Pi_1 \cdots \Pi_n)$, by

$$h((q^*, i^*, j^*)) = \begin{cases} (q', i^*, j^*) & \text{if } q^* = q \text{ and } j^* \geq j' \\ (q^*, i^*, j^*) & \text{otherwise.} \end{cases}$$

Let $LS = RD(\Pi_1 \cdots \Pi_{j'-1}\Pi_{new}\Pi'_{j'} \cdots \Pi'_n)$ and $RS = RD(\Pi_1 \cdots \Pi_n)$. It is straightforward to verify that $LS \subseteq h(RS)$. Thus, $|LS| \leq |h(RS)|$. Now, $(q', i', j')$ is in $h(RS)$, but not in $LS$. Therefore, $|LS| < |h(RS)|$. Furthermore, $h$ is one-to-one. Hence, $|h(RS)| = |RS|$, and Claim B is proven.

Repeating the transformation (defined just prior to Claim A) a finite number of times, one ultimately obtains a $p$-decomposition $\Pi^*_1 \cdots \Pi^*_k$ of $\Pi$ with no reverse dependency. Since each $\Pi_i$ is an SCC of itself, clearly each $\Pi^*_j$ is an SCC of itself. Thus, $\Pi^*_1 \cdots \Pi^*_k$ is a normal $p$-decomposition of $\Pi$. By the construction, for $1 \leq j \leq k$, either $\Pi^*_j$ has the form $\{q_1(X,Y) :- q_2(X,Y).\}$; or there is an $i$ $(1 \leq i \leq n)$ such that $\Pi^*_j$ is obtained from $\Pi_i$ by a renaming of all occurrences of some IDB predicates of $\Pi$. This completes the proof of the lemma. □

Although not needed in the sequel, the following is of interest: For each sequence $\Pi_1 \cdots \Pi_n$ and predicate $p$, there is a program $\Pi$ which is $p$-equivalent to $\Pi_1 \cdots \Pi_n$ over $EDB(\cup_{i=1}^n \Pi_i)$. In comparison, there is no such program in the uniform sense [4] (where IDB predicates are allowed to carry input as well).

We now illustrate the construction in Lemma 3.2.

*Example 3.1.* Consider the $p_1$-decomposition $\Pi_1 \cdots \Pi_7$ of $\Pi$, where $\Pi_i = \{r_i\}$ for $1 \leq i \leq 7$ and $\Pi = \{r'_1, r'_2, r', r'_3, r_4, r_5, r_6, r_7\}$,

$$\begin{aligned}
r_1&: p_1(X,Z) :- p_1(X,Y), p_4(Y,Z).\\
r_2&: p_1(X,Z) :- p_4(X,Z).\\
r_3&: p_4(X,Z) :- p_3(X,Y_1), b_2(Y_1,Y_2), a_2(Y_2,Z).\\
r_4&: p_3(X,Z) :- p_3(X,Y), b_2(Y,Z).\\
r_5&: p_3(X,Z) :- p_4(X,Y_1), b_1(Y_1,Y_2), a_1(Y_2,Z).\\
r_6&: p_4(X,Z) :- p_4(X,Y), b_1(Y,Z).\\
r_7&: p_4(X,Z) :- a_0(X,Z).\\
r'_1&: p_1(X,Z) :- p_1(X,Y), p_2(Y,Z).\\
r'_2&: p_1(X,Z) :- p_2(X,Z).\\
r'&: p_1(X,Z) :- p_4(X,Z).\\
r'_3&: p_2(X,Z) :- p_3(X,Y_1), b_2(Y_1,Y_2), a_2(Y_2,Z).
\end{aligned}$$

There are two reverse dependencies in the given decomposition, namely, $(p_4, 3, 5)$

[3] For each finite set $S$, $|S|$ denotes the number of elements in $S$.

and $(p_4, 3, 6)$. In order to remove the reverse dependency $(p_4, 3, 5)$, we follow the construction in Lemma 3.2 and obtain a new $p_1$-decomposition $\Pi_1 \cdots \Pi_4 \Pi_{41} \Pi_{51} \Pi_{61} \Pi_{71}$, where

$$\begin{aligned}
\Pi_{41}: p_4(X, Z) &:- p_{41}(X, Z).\\
\Pi_{51}: p_3(X, Z) &:- p_{41}(X, Y_1), b_1(Y_1, Y_2), a_1(Y_2, Z).\\
\Pi_{61}: p_{41}(X, Z) &:- p_{41}(X, Y), b_1(Y, Z).\\
\Pi_{71}: p_{41}(X, Z) &:- a_0(X, Z).
\end{aligned}$$

Here, $\Pi_{51}$, $\Pi_{61}$, and $\Pi_{71}$ are the results of renaming the predicate $p_4$ by $p_{41}$ in $\Pi_5$, $\Pi_6$, and $\Pi_7$, respectively, and $\Pi_{41}$ is the new "transition" program. This transformation also removes the reverse dependency $(p_4, 3, 6)$. Since no reverse dependency now exists, $\Pi_1 \cdots \Pi_4 \Pi_{41} \Pi_{51} \Pi_{61} \Pi_{71}$ is a normal $p_1$-decomposition of $\Pi$.

To establish results for decomposability into decompositions with 1-rule components, we shall consider more general decompositions in which each component "defines" exactly one predicate.

*Definition 3.5.* A program $\Pi$ is called *unihead* if $IDB(\Pi)$ is a singleton, that is, if all the predicates appearing in the rule heads are the same.

Obviously, 1-rule programs are unihead programs.

For context-free languages and grammars, we need the parallel for productions of the terms "depends on," "SCC," and "unihead." The definitions are as above by replacing "program $\Pi$" with "context-free grammar $G$," "rule" with "production," and "IDB predicate" with "variable," respectively. Using these notions, we now define "$m$-sequentiality." (The concept of "sequentiality" was first studied in [8]. The original motivation was to solve "equations" sequentially and one at a time.)

*Definition 3.6.* A context-free grammar $G = (V, \Sigma, P, p_1)$ is *sequential* if $V = \{p_1, \ldots, p_n\}$ such that $j \geq i$ for each production $p_i \to up_jv \in P$. And $G$ is *$m$-sequential*, $m$ an integer, if it is sequential and there are at most $m$ productions in each SCC of $G$.

It is worth noting that each SCC of a sequential grammar is unihead. Furthermore, if $p_i$ is the variable in the head of each production in an SCC containing at least two productions, then $p_i$ also occurs in the body of each production in that SCC.

Suppose $\Pi_1 \cdots \Pi_n$ is a normal sequence where each $\Pi_i$ is unihead. Then it is easily seen that $G_{\cup_{j=1}^{n} \Pi_j, p}$ is sequential for each predicate $p$.

The final technical lemma needed for the main result of the section characterizes the equivalence of programs in terms of the equality of the associated languages.

*Lemma 3.3. Two programs $\Pi_1$ and $\Pi_2$ are $p$-equivalent iff $L(G_{\Pi_1, p}) = L(G_{\Pi_2, p})$.*

PROOF. We define the *fringe* of a derivation tree as the sequence of its leaves. Let $\Pi_1$ and $\Pi_2$ be programs and $p$ a predicate. For the "if," we use the following result [29]:

(†) For each $i$, $1 \leq i \leq 2$, $q_1(x_1, x_2), q_2(x_2, x_3), \ldots, q_n(x_n, x_{n+1})$ is the fringe of a $\Pi_i$-derivation tree for $p(x_1, x_{n+1})$ iff $q_1 q_2 \cdots q_n$ is in $L(G_{\Pi_i,p})$. [It is understood that $x_1, \ldots$, and $x_{n+1}$ are mutually distinct constants.]

Note that the fringe in the statement of (†) is the fringe of a "most general derivation tree." (That is, the fringe of any other derivation tree is the "image" of some "most general derivation" under some mapping from constants onto constants.) Hence, all other derivation trees can be obtained by renaming the constants in the tree. The details are omitted here.

For the "only if," suppose that $L(G_{\Pi_1,p}) \neq L(G_{\Pi_2,p})$, say $L(G_{\Pi_1,p}) \not\subseteq L(G_{\Pi_2,p})$. Then there exists $q_1 \cdots q_n$ in $L(G_{\Pi_1,p}) - L(G_{\Pi_2,p})$. Let $x_1, \ldots, x_{n+1}$ be distinct constants, and $I = \{q_i(x_i, x_{i+1}) \mid 1 \leq i \leq n\}$. By (†), $p(x_1, x_{n+1})$ is in $T^{\omega}_{\Pi_1}(I)$, but not in $T^{\omega}_{\Pi_2}(I)$. Hence, $\Pi_1$ and $\Pi_2$ are not $p$-equivalent. □

We transfer the notions of limiting mapping and $\mathcal{P}_\Pi$ (left-)linear to grammars. For each limiting mapping $\mathcal{P}$, a grammar $G$ is called $\mathcal{P}$ *(left-)linear* if it is $\mathcal{P}_G$ (left-)linear.

We are now ready for the main result of the section.

*Theorem 3.1. Let $\mathcal{P}$ be an arbitrary limiting mapping. Then a program $\Pi$ has a $\mathcal{P}$ (left-)linear p-decomposition with 1-rule components*[4] *iff $L(G_{\Pi,p})$ is generated by a 1-sequential $\mathcal{P}$ (left-)linear grammar. More generally, for each integer $m \geq 1$, $\Pi$ has a $\mathcal{P}$ (left-)linear p-decomposition with m-rule unihead components iff $L(G_{\Pi,p})$ is generated by an m-sequential $\mathcal{P}$ (left-)linear grammar.*

PROOF. Let $m \geq 1$ be an integer, $\Pi$ a program, and $p$ a predicate. We only consider the $\mathcal{P}$-linear case, the left-linear case being similar. For the "if," suppose $L(G_{\Pi,p})$ is generated by an $m$-sequential $\mathcal{P}$-linear grammar $G = (V, \Sigma, P, p)$. Without loss of generality, we may suppose that each variable of $V$ appears as the head of a production in $P$, and each terminal of $\Sigma$ occurs in a production in $P$. Let

$$\Pi' = \{q(X, Z) :- q_1(X, Y_1), q_2(Y_1, Y_2), \ldots, q_k(Y_{k-1}, Z). \mid q \to q_1 \cdots q_k \text{ in } P, \text{ and } X, Y_1, \ldots, Y_{k-1}, Z \text{ are distinct variables}\}.$$

Clearly, $G_{\Pi',p} = G$. Thus, $L(G_{\Pi',p}) = L(G_{\Pi,p})$, and $G_{\Pi',p}$ is $m$-sequential and $\mathcal{P}$-linear. Hence, $\Pi'$ is $\mathcal{P}_{\Pi'}$-linear, each SCC of $\Pi'$ contains at most $m$ rules, and (as noted earlier) is unihead. By Lemma 3.1, $\Pi'$ has a $\mathcal{P}$-linear $p$-decomposition with $m$-rule unihead components. By Lemma 3.3, so does $\Pi$.

For the "only if," suppose there exists a $\mathcal{P}$-linear $p$-decomposition $\Pi_1 \cdots \Pi_n$ of $\Pi$ with $m$-rule unihead components. By Lemma 3.2, there exists a normal $\mathcal{P}$-linear $p$-decomposition $\Pi_1^* \cdots \Pi_k^*$, with $m$-rule unihead components, of $\Pi$. Therefore, each SCC of $\Pi^* = \Pi_1^* \cup \cdots \cup \Pi_k^*$ is some $\Pi_i^*$. Since $\Pi_1^* \cdots \Pi_k^*$ is normal, each SCC of $G_{\Pi^*,p}$ is a set of productions corresponding to some $\Pi_i^*$. Hence, $G_{\Pi^*,p}$ is $m$-sequential and $\mathcal{P}$-linear. By Lemma 3.3, $L(G_{\Pi^*,p}) = L(G_{\Pi,p})$. Thus, $L(G_{\Pi,p})$ is generated by an $m$-sequential $\mathcal{P}$-linear grammar. □

[4]That is, each component program in the decomposition has at most one rule. Similar phrases have obvious similar meanings.

Consider the following simple program $\Pi$:

$$p(X,Z) \text{ :- } a(X,Y_1), p(Y_1,Y_2), d(Y_2,Z).$$
$$p(X,Z) \text{ :- } b(X,Y_1), p(Y_1,Y_2), d(Y_2,Z).$$
$$p(X,Z) \text{ :- } c(X,Z).$$

Clearly, $L(G_{\Pi,p}) = \{x_1 \cdots x_n c d^n \mid 0 \leq n, x_i \in \{a,b\} \text{ for each } i, 1 \leq i \leq n\}$. Furthermore, $L(G_{\Pi,p}) = L(G')$, where $G'$ is the 1-sequential grammar $(\{p,p'\}, \{a,b,c,d\}, \{p \rightarrow p'pd, p' \rightarrow a, p' \rightarrow b, p \rightarrow c\}, p)$. Let $\mathcal{P}$ be defined by $\mathcal{P}(\Pi', r) = \emptyset$ for all $\Pi'$ and $r$. By Theorem 3.1, $\Pi$ has a $\mathcal{P}$ (left-)linear $p$-decomposition with 1-rule components. In fact, $\Pi_1\Pi_2\Pi_3\Pi_4$ is such a $p$-decomposition, where

$$\Pi_1\text{: } p(X,Z) \text{ :- } p'(X,Y_1), p(Y_1,Y_2), d(Y_2,Z).$$
$$\Pi_2\text{: } p'(X,Z) \text{ :- } a(X,Z).$$
$$\Pi_3\text{: } p'(X,Z) \text{ :- } b(X,Z).$$
$$\Pi_4\text{: } p(X,Z) \text{ :- } c(X,Z).$$

In later sections, we shall present more involved examples.

We now turn to the decidability issue regarding the existence of $\mathcal{P}$ (left-)linear decompositions with 1-rule components. Obviously, we should expect to have an undecidability result for arbitrary $\mathcal{P}$, where the undecidability is due to the "non-recursive computability" of $\mathcal{P}$. Thus, it is only meaningful to consider $\mathcal{P}$ with special forms. As it turns out, for some $\mathcal{P}$, we have decidability, while for others, undecidability. We shall give an undecidability result for a degenerate $\mathcal{P}$ below, and return to this issue for other special $\mathcal{P}$s in the next section.

We first need some notation and a technical lemma.

*Notation* For each context-free language $L$ and word $v$, let $Suf_v(L)$ denote $\{w \neq \Lambda \mid vw \in L\}$.

The technical lemma is the following:

*Lemma 3.4. For each $\Sigma$ with at least two elements, let $S$ be a function from the family of context-free languages over $\Sigma$ into $\{true, false\}$ such that*

a. *$S(\Sigma^+) = true$, and*
b. *$\{Suf_v(L) \mid v \in \Sigma^+, S(L) = true\}$ is a proper[5] subfamily of the context-free languages over $\Sigma$.*

*Then it is undecidable whether $S(L(G)) = true$ for arbitrary context-free grammars $G$.*

PROOF. Let $S^*$ be the function defined (for each extended context-free language $L$ over $\Sigma$; see Section 2.3) by $S^*(L) = true$ if $S(L - \{\Lambda\}) = true$ and $S^*(L) = false$ otherwise. Let $Suf_v^*(L) = \{w \mid vw \in L\}$ for each $L$ as above and each $v \in \Sigma^+$. (Note that $w$ can be $\Lambda$ in this definition.) From (a) and (b), it follows that (i) $S^*(\Sigma^*) = true$, and (ii) $\{Suf_v^*(L) \mid v \in \Sigma^+, S^*(L) = true\}$ is a proper subfamily

[5] A family $\mathcal{L}_1$ is called a *proper subfamily* of a family $\mathcal{L}_2$ if $\mathcal{L}_1 \subset \mathcal{L}_2$, i.e., each member of $\mathcal{L}_1$ is a member of $\mathcal{L}_2$, but not vice versa.

of the extended context-free languages. By [14, Theorem 2.1], it is undecidable if $S^*(L(G)) = true$ for an arbitrary extended context-free grammar $G$. Since a $\Lambda$-free context-free grammar can be effectively constructed for $L(G) - \{\Lambda\}$ for each $G$ as above [12], it follows that it is undecidable if $S(L(G)) = true$ for an arbitrary $\Lambda$-free context-free grammar $G$. □

We are now able to present the undecidability result. (This should not be confused with the result in [9] which states that it is undecidable whether or not a context-free language is sequential.)

*Theorem 3.2. Let $\mathcal{P}$ be the limiting mapping defined by $\mathcal{P}(\Pi, r) = \emptyset$ for all $\Pi$ and $r$. Then it is undecidable whether $\Pi$ has a $\mathcal{P}$ (left-)linear p-decomposition with 1-rule components for all arbitrary predicates $p$ and programs $\Pi$.*

PROOF. Clearly, we can effectively construct $G_{\Pi,p}$ for all programs $\Pi$ and predicates $p$. By Theorem 3.1, it suffices to show that it is undecidable whether an arbitrary context-free language is 1-sequential. It suffices to show that it is undecidable whether an arbitrary context-free language with at least three terminals is 1-sequential. Let $\Sigma$ be an arbitrary alphabet of at least three elements. Let $S$ be the function from the family of context-free languages over $\Sigma$ into $\{true, false\}$ such that $S(L) = true$ iff $L$ is 1-sequential. Clearly, $S(\Sigma^+) = true$. [Indeed, let $G_0 = (\{p_1, p_2\}, \Sigma, P, p_1)$ with $P = \{p_1 \rightarrow p_1p_2, p_1 \rightarrow a, p_2 \rightarrow a \mid a \in \Sigma\}$. Then $L(G_0) = \Sigma^+$ and $G_0$ is 1-sequential.] By Proposition A of the Appendix, $\{Suf_v(L) \mid v \in \Sigma^+, L \subseteq \Sigma^+, S(L) = true\}$ is a proper subfamily of the context-free languages over $\Sigma$. By Lemma 3.4, the condition of $S(L(G)) = true$ for an arbitrary context-free grammar $G$ is undecidable. □

REMARK 3.1. The following orthogonal result was shown in [24]: (*) It is undecidable to determine for an "arbitrary general Horn-clause program" $\Pi = \{r_1, r_2\}$ if $\Pi$ $p$-decomposes into $\{r_2\}\{r_1\}$ for a given predicate $p$. The study in [24] in general, and (*) in particular, uses predicates with big arities and disallows new rules in decompositions.

The remainder of this section shows that (i) arbitrarily large "size-prime" (or "nondecomposable") programs exist, and (ii) decomposability into 1-rule programs is different from decomposability into 2-rule programs. To this end, we need two concepts and one preliminary result.

The first concept is:

*Definition 3.7.* A program $\Pi'$ is said to be an *initialization program* for a program $\Pi$ if, for each rule $r$ in $\Pi'$, the body of $r$ does not contain any predicate in $IDB(\Pi \cup \Pi')$.

Initialization programs are necessary for defining "size primes." Indeed, suppose a program $\Pi$ contains an initialization component and has two or more rules. Then $\Pi$ is not an SCC, and is thus decomposable into programs that are SCCs of $\Pi$ by Lemma 3.1.

The preliminary result, which is of interest in its own right, is the following:

*Lemma 3.5.* *Let $\mathcal{P}$ be a limiting mapping and for each integer $m \geq 2$, let $\Pi$ be the $m$-rule unihead program*

$$\{p(X,Z) :- a_i(X,Y_1), p(Y_1,Y_2), a_i(Y_2,Z). \mid 1 \leq i \leq m\}$$

*where $a_1, \ldots, a_m$ are distinct predicates. Then* (i) *$\Pi$ is $\mathcal{P}_\Pi$-linear;* (ii) *for each initialization program $\Pi''$ for $\Pi$, where $\Pi''$ is unihead and $m$-rule, $\Pi\Pi''$ is a $\mathcal{P}$-linear $p$-decomposition of $\Pi \cup \Pi''$ with $m$-rule unihead components; and* (iii) *there exists an initialization $\Pi'$ for $\Pi$, where $\Pi'$ is 1-rule, such that $\Pi \cup \Pi'$ has no $\mathcal{P}$-linear $p$-decomposition, of the form $\Pi_1 \cdots \Pi_k \Pi'$, with $(m-1)$-rule unihead components.*

PROOF. Consider (i). Let $\mathcal{P}'$ be the limiting mapping defined by $\mathcal{P}'(\Pi_0, r') = IDB(\Pi_0)$ for each $\Pi_0$ and $r'$. Then $\Pi$ is $\mathcal{P}'_\Pi$-linear. As noted just prior to Lemma 3.1, $\Pi$ is $\mathcal{P}''_\Pi$-linear for each limiting mapping $\mathcal{P}''$. Thus, (i) is proven. Clearly, (ii) holds. Now, consider (iii). Let $\Pi' = \{p(X,Z) :- c(X,Z).\}$, $c$ a new predicate. Let $\mathcal{P}''$ be defined by $\mathcal{P}''(\Pi_0, r') = \emptyset$ for all $\Pi_0$ and $r'$. Assume $\Pi \cup \Pi'$ has a $\mathcal{P}$-linear $p$-decomposition $\Pi_1 \cdots \Pi_k \Pi'$ with $(m-1)$-rule unihead components. Since $\mathcal{P}''(\Pi', r) = \emptyset \subseteq \mathcal{P}(\Pi', r)$ for each $\Pi'$ and $r$, each $\mathcal{P}$-linear program is clearly $\mathcal{P}''$-linear. Therefore, $\Pi_1 \cdots \Pi_k \Pi'$ is a $\mathcal{P}''$-linear $p$-decomposition with $(m-1)$-rule unihead components of $\Pi \cup \Pi'$. Let $L = L(G_{\Pi \cup \Pi', p})$. By Theorem 3.1, $L$ is $(m-1)$-sequential. Note that $L$ is $L_0$, where $L_0$ is as defined in the Appendix. Clearly, $L_0 = Suf_\Lambda(L)$. But this contradicts Proposition A of the Appendix. Hence, (iii) holds. □

In [7], the notion of "prime" Datalog programs was introduced and examined. Our second concept is a variation of that idea.

*Definition 3.8.* For each limiting mapping $\mathcal{P}$ and predicate $p$, a program $\Pi$ is said to be a $\mathcal{P}$ *(left-)linear size prime with respect to $p$* if (i) $\Pi$ is $\mathcal{P}_\Pi$ (left-)linear and unihead, and (ii) there is an initialization program $\Pi'$ such that $\Pi \cup \Pi'$ has no $\mathcal{P}$ (left-)linear $p$-decomposition of the form $\Pi_1 \cdots \Pi_n \Pi'$, where each $\Pi_i$ $(1 \leq i \leq n)$ is unihead and has fewer rules than $\Pi$.

For each limiting mapping $\mathcal{P}$ and predicate $p$, it follows from Lemma 3.1 that only SCCs can be $\mathcal{P}$ (left-)linear size primes with respect to $p$. It will be seen in the next section that there exist (i) some limiting mapping $\mathcal{P}$, (ii) a predicate $p$, and (iii) a program $\Pi$ which is an SCC such that $\Pi$ is not a $\mathcal{P}$ (left-)linear size prime with respect to $p$.

Our first result regarding the existence of size-primes follows immediately from Lemma 3.5:

*Theorem 3.3.* *For each limiting mapping $\mathcal{P}$ and predicate $p$, there exist arbitrarily large $\mathcal{P}$-linear size-prime programs with respect to $p$.*

Theorem 3.3 deals with the existence of $\mathcal{P}$-linear size-prime programs for arbitrary $\mathcal{P}$. We shall see (Corollary 4.2) that the counterpart of this theorem does not hold for $\mathcal{P}$ left-linear size primes in general.

Our last result of this section follows from Lemma 3.5 by letting $m = 2$. It concerns the difference between decompositions into 1-rule and decompositions into 2-rule programs.

*Theorem 3.4. For each limiting mapping $\mathcal{P}$, each $\mathcal{P}$-linear p-decomposition with 1-rule components is a $\mathcal{P}$-linear p-decomposition with 2-rule unihead components but not vice versa.*

## 4. HEAD (LEFT-)LINEAR DECOMPOSITIONS

We now consider a type of $\mathcal{P}$ (left-)linear decomposition, called "head (left-)linear," induced by a particular limiting mapping $\mathcal{P}^h$. We then examine two special subfamilies of these decompositions, called "recursion-dependent (left-)linear" and "recursion-one (left-)linear," each defined by a limiting mapping more restrictive than $\mathcal{P}^h$. Head (left-)linear emphasizes the (left) linearity in a rule body of the head predicate of that rule. The two subfamilies guarantee a bounded number of IDB predicate occurrences at each step of a derivation. The main results in this section are characterizations relating these decompositions for programs with properties of the programs' associated languages.

### *4.1. Head (Left-)Linear Decompositions*

In this subsection, we examine the existence of head (left-)linear decompositions with 1-rule components. In such decompositions, each 1-rule program is (left) linear in the traditional sense.

We start with the definition of "head (left-)linear."

*Definition 4.1.* Let $\mathcal{P}^h$ be the limiting mapping defined (for all $\Pi$ and $r$) by (i) $\mathcal{P}^h(\Pi, r) = IDB(\{r\})$ if $r$ is a variable renaming of a rule in $\Pi$ and the head predicate of $r$ appears in its body, and (ii) $\mathcal{P}^h(\Pi, r) = \emptyset$ otherwise. A decomposition $\Pi_1 \cdots \Pi_n$ is called *head (left-)linear* if it is $\mathcal{P}^h$ (left-)linear. A grammar $G = (V, \Sigma, P, p)$ is called *head (left-)linear* if it is $\mathcal{P}^h_G$ (left-)linear, where $\mathcal{P}^h$ is transferred to grammars in the obvious way. That is, $G$ is head (left-)linear if, for each production $p' \to w$ in $P$, $p'$ can occur at most once in $w$ (and only as the leftmost symbol of $w$).

For example, the program

$$\begin{aligned} p(X,Y) &:- a(X,Z_1),\ p(Z_1,Z_2),\ p_1(Z_2,Y). \\ p_1(X,Y) &:- a(X,Y). \end{aligned}$$

is head linear. Note that the IDB predicates $p$ and $p_1$ both occur in the first rule. But the important fact is that $p$ only occurs once in the body of that rule. On the other hand, the program

$$\begin{aligned} p(X,Y) &:- a(X,Z_1),\ p(Z_1,Z_2),\ p(Z_2,Y). \\ p(X,Y) &:- a(X,Y). \end{aligned}$$

is not head linear since $p$ occurs twice in the body of the first rule.

Clearly, head linear emphasizes only the linearity of the head predicate in the body of each rule under consideration. Also, head (left-)linear is equivalent to traditional (left-)linear for 1-rule programs.

By Theorem 3.1, we obtain the following result.

*Theorem 4.1. A program $\Pi$ has a head (left-)linear p-decomposition with 1-rule components iff $L(G_{\Pi,p})$ is generated by a 1-sequential head (left-)linear grammar.*

We now turn to characterizing a large class of programs possessing head (left-)-linear decompositions with 1-rule components. To this end and for later usage, we need the notions of "Kleene derivation" and "regular" languages.

*Definition 4.2.* A $\{\cup, \bullet, +\}$-*derivation* for a language $L$ over $\Sigma$ is a sequence $L_1, \ldots, L_n$ of languages such that $L_n = L$ and each $L_i$ has one of the following forms: $\emptyset$, $\{a\}$, $L_j \cup L_m$, $L_j \bullet L_m$, or $L_j^+$, where $a \in \Sigma$ and $1 \leq j, m < i$. A set $L$ over $\Sigma$ is called *regular* if there exists a $\{\cup, \bullet, +\}$-derivation for $L$.

Note that each $L_i$ in a $\{\cup, \bullet, +\}$-derivation is $\Lambda$-free. Regular languages as defined above clearly coincide with regular languages defined by "left-linear" grammars.

The second result of this subsection deals only with the left-linear case.

*Theorem 4.2. A program $\Pi$ has a head left-linear p-decomposition with 1-rule components iff $L(G_{\Pi,p})$ is regular.*

PROOF. Let $\Pi$ be a program and $p$ a predicate. For the "if," suppose $L(G_{\Pi,p})$ is regular. Then there exists a $\{\cup, \bullet, +\}$-derivation $L_1, \ldots, L_n$ for $L(G_{\Pi,p})$. Let $p_n = p$ and let $p_1, \ldots, p_{n-1}$ be new predicates. For each $1 \leq i \leq n$, let $\Pi_i$ be defined as follows:

(a) If $L_i = \emptyset$, then $\Pi_i = \emptyset$.

(b) If $L_i = \{a\}$, where $a$ is in $\Sigma$, then $\Pi_i$ is

$$p_i(X, Z) \text{ :- } a(X, Z).$$

(c) If $L_i = L_j \cup L_m$, then $\Pi_i$ is

$$\begin{aligned} p_i(X, Z) &\text{ :- } p_j(X, Z). \\ p_i(X, Z) &\text{ :- } p_m(X, Z). \end{aligned}$$

(d) If $L_i = L_j \bullet L_m$, then $\Pi_i$ is

$$p_i(X, Z) \text{ :- } p_j(X, Y), p_m(Y, Z).$$

(e) If $L_i = L_j^+$, then $\Pi_i$ is

$$\begin{aligned} p_i(X, Z) &\text{ :- } p_i(X, Y), p_j(Y, Z). \\ p_i(X, Z) &\text{ :- } p_j(X, Z). \end{aligned}$$

Let $\Pi' = \cup_{i=1}^n \Pi_i$. Clearly, $L(G_{\Pi',p}) = L(G_{\Pi,p})$, $\Pi'$ is head left-linear, and each SCC of $\Pi'$ consists of one rule. By Lemma 3.3, $\Pi$ and $\Pi'$ are $p$-equivalent. By

Lemma 3.1, $\Pi'$ has a head left-linear $p$-decomposition $\Pi'_1 \cdots \Pi'_k$ with 1-rule components. It follows that $\Pi'_1 \cdots \Pi'_k$ is a head left-linear $p$-decomposition of $\Pi$ with 1-rule components. Thus the "if" is proven.

For the "only if," suppose $\Pi$ has a head left-linear $p$-decomposition with 1-rule components. By Theorem 4.1, $L(G_{\Pi,p})$ is generated by a head left-linear 1-sequential grammar. It thus suffices to show that

(*) $L(G)$ is regular for each head left-linear 1-sequential grammar $G$.

To this end, let $G = (\{p_1, \ldots, p_l\}, \Sigma, P, p_1)$ be a head left-linear 1-sequential grammar. Suppose there exists a rule of the form $p_1 \to p_1 w$, where $w \in (\Sigma \cup \{p_2, \ldots, p_l\})^+$. (The case when no such rule exists is similar and omitted.) Since $G$ is 1-sequential, this rule is unique. Let $p_1 \to u_i$, $1 \leq i \leq n$, be the other rules with $p_1$ as head. For $l = 1$, (*) holds because $u_1, \ldots, u_n$ and $w$ are terminal words. Continuing by induction, suppose $k$ is a positive integer such that (*) holds for each $l \leq k$. Consider $l = k + 1$. Let $j$ $(1 < j \leq l)$ be fixed, and let $G_j = (\{p_j, \ldots, p_l\}, \Sigma, P_j, p_j)$, where $P_j$ consists of all productions of the form $p_i \to v$ in $P$ such that $v \in (\Sigma \cup V)^*$ and $i \geq j$. Clearly, $G_j$ has at most $k$ variables. Since $G$ is 1-sequential and head left-linear, so is $G_j$. By induction, $L(G_j)$ is regular. Let $G_1 = (\{p_1\}, \Sigma \cup \{p_2, \ldots, p_l\}, \{p_1 \to p_1 w, p_1 \to u_1, \ldots, p_1 \to u_n\}, p_1)$. By induction, $L(G_1)$ is regular. Let $\sigma$ be the language substitution[6] defined by $\sigma(p_i) = L(G_i)$ for $1 < i \leq l$ and $\sigma(a) = \{a\}$ for $a \in \Sigma$. Obviously, $\sigma(L(G_1)) = L(G)$. Since the family of regular languages is closed under language substitution by regular languages [12], (*) holds, i.e., $L(G)$ is regular as desired. □

Since it is undecidable whether an arbitrary context-free language is regular [12], the above theorem yields:

*Corollary 4.1. It is undecidable whether an arbitrary program has a head left-linear p-decomposition with 1-rule components.*

We now use Theorem 4.2 to obtain a corollary on size primes. First, though, we need the following concept: A context-free grammar $(V, \Sigma, P, p)$ is called *(left-)linear* if, for each production of the form $p' \to w$ in $P$, there is at most one occurrence of variables from $V$ in $w$ (and only as the leftmost symbol in $w$).

The corollary on size primes is:

*Corollary 4.2. A nonempty unihead program is a head left-linear size prime iff it has exactly one rule. Consequently,* (i) *there are no arbitrarily large head left-linear size-primes, and* (ii) *head left-linear decompositions with 1-rule components are identical to head (left-)linear decompositions with 2-rule unihead components.*

PROOF. Clearly, each 1-rule unihead program is a head left-linear size prime. For the "only if," suppose $\Pi$ is a unihead program with two or more rules. Let $p$ be

[6] A *language substitution* over $\Sigma$ is a mapping $\sigma$ from $\Sigma^*$ to languages over $\Sigma$ such that $\sigma(\Lambda) = \{\Lambda\}$ and $\sigma(a_1 a_2 \cdots a_k) = \sigma(a_1)\sigma(a_2)\cdots\sigma(a_k)$ for all $k \geq 1$ and $a_1, a_2, \ldots, a_k$ in $\Sigma$. For each set $L$ of words over $\Sigma$, $\sigma(L)$ is defined as $\cup_{w \in L} \sigma(w)$.

the head predicate of the rules in $\Pi$, and let $\Pi'$ be an arbitrary initialization for $\Pi$. Then, $G_{\Pi\cup\Pi',p}$ is left-linear. Thus, as mentioned prior to Theorem 4.2, $L(G_{\Pi\cup\Pi',p})$ is regular. By Theorem 4.2, there is a head left-linear $p$-decomposition $\Pi_1 \cdots \Pi_k$ for $\Pi \cup \Pi'$. By applying an appropriate predicate renaming to $\Pi_1 \cdots \Pi_k$ if necessary, we may assume that no IDB predicate of $\Pi'$ except $p$ is used in $\Pi_1 \cdots \Pi_k$. It then follows that $\Pi_1 \cdots \Pi_k \Pi'$ is also a head left-linear $p$-decomposition for $\Pi \cup \Pi'$. Since each $\Pi_i$ has only one rule, $\Pi$ cannot be a head left-linear size prime. □

*Example 4.1.* To illustrate the "if" in Theorem 4.2, consider the following program $\Pi$:

$$
\begin{aligned}
p(X,Z) &:- p_2(X,Y_1), b_2(Y_1,Y_2), a_2(Y_2,Z).\\
p_2(X,Z) &:- p_2(X,Y), b_2(Y,Z).\\
p_2(X,Z) &:- p_1(X,Y_1), b_1(Y_1,Y_2), a_1(Y_2,Z).\\
p_1(X,Z) &:- p_1(X,Y), b_1(Y,Z).\\
p_1(X,Z) &:- p(X,Y), a_0(Y,Z).\\
p_1(X,Z) &:- a_0(X,Z).
\end{aligned}
$$

Then $L(G_{\Pi,p})$ is $(\{a_0\}\{b_1\}^+\{a_1\}\{b_2\}^+\{a_2\})^+$, and thus regular. One $\{\cup,\bullet,+\}$-derivation for $L(G_{\Pi,p})$ is: $\{a_0\}$, $\{b_1\}$, $\{a_1\}$, $\{b_2\}$, $\{a_2\}$, $\{b_1\}^+$, $\{b_2\}^+$, $\{b_2\}^+\{a_2\}$, $\{a_1\}\{b_2\}^+\{a_2\}$, $\{b_1\}^+\{a_1\}\{b_2\}^+\{a_2\}$, $\{a_0\}\{b_1\}^+$ $\{a_1\}\{b_2\}^+$ $\{a_2\}$, $(\{a_0\}\{b_1\}^+$ $\{a_1\}$ $\{b_2\}^+$ $\{a_2\})^+$. Let $\Pi_1$, ..., $\Pi_{15}$ be defined as follows:

$$
\begin{aligned}
\Pi_1:\ & p_1(X,Z) &:-\ & a_0(X,Z).\\
\Pi_2:\ & p_2(X,Z) &:-\ & b_1(X,Z).\\
\Pi_3:\ & p_3(X,Z) &:-\ & a_1(X,Z).\\
\Pi_4:\ & p_4(X,Z) &:-\ & b_2(X,Z).\\
\Pi_5:\ & p_5(X,Z) &:-\ & a_2(X,Z).\\
\Pi_6:\ & p_6(X,Z) &:-\ & p_2(X,Z).\\
\Pi_7:\ & p_6(X,Z) &:-\ & p_6(X,Y), p_2(Y,Z).\\
\Pi_8:\ & p_7(X,Z) &:-\ & p_4(X,Z).\\
\Pi_9:\ & p_7(X,Z) &:-\ & p_7(X,Y), p_4(Y,Z).\\
\Pi_{10}:\ & p_8(X,Z) &:-\ & p_7(X,Y), p_5(Y,Z).\\
\Pi_{11}:\ & p_9(X,Z) &:-\ & p_3(X,Y), p_8(Y,Z).\\
\Pi_{12}:\ & p_{10}(X,Z) &:-\ & p_6(X,Y), p_9(Y,Z).\\
\Pi_{13}:\ & p_{11}(X,Z) &:-\ & p_1(X,Y), p_{10}(Y,Z).\\
\Pi_{14}:\ & p(X,Z) &:-\ & p_{11}(X,Z).\\
\Pi_{15}:\ & p(X,Z) &:-\ & p(X,Y), p_{11}(Y,Z).
\end{aligned}
$$

Then $\Pi_{15} \cdots \Pi_1$ (the number is reversed for ease of illustration) is a head left-linear $p$-decomposition of $\Pi$ with 1-rule components as constructed in the proof. Note that the decomposition can be simplified into ten rules by removing the first five rules and adjusting the other rules appropriately.

In passing, we note the following two points:

1. Although regular languages are known to have left-linear grammars, Theorem 4.2 also shows that regular languages have 1-sequential head left-linear grammars.
2. For each $\{\cup, \bullet, +\}$-derivation $L_1, \ldots, L_n$, Theorem 4.2 yields a head left-linear $p$-decomposition with a sequence of at most $2n$ rules.

*4.2. Recursion-Dependent (Left-)Linear Decompositions*

Head (left-)linear decompositions with 1-rule components are clearly preferable to general decompositions with 1-rule components. However, even for an arbitrary head (left-)linear decomposition $\{r_1\} \cdots \{r_m\}$, there is still no limit on the number of occurrences in each $r_i$ of predicates which depend on "recursive" predicates of $\{r_j \mid i < j \leq m\}$. This deficiency is overcome here and in the next subsection by considering decompositions which are linear in a stronger sense.

We begin with several recursion-related concepts.

*Definition 4.3.* A rule is called *recursive* in a program if its head predicate occurs in its body or it is in an SCC of two or more rules. A predicate is called *recursive* in a program if it is the head predicate of a recursive rule.

Roughly speaking, "recursion-dependent" predicates are those which can lead to recursive predicates in derivations. Specifically, we have:

*Definition 4.4.* Suppose $\Pi$ is a program. For all $p$ and $p'$ in $IDB(\Pi)$, $p$ is said to *depend on* $p'$ if either there is a rule in $\Pi$ with $p$ in its head and $p'$ in its body, or there is a $p''$ in $IDB(\Pi)$ such that $p$ depends on $p''$ and $p''$ depends on $p'$. A predicate $p$ in $IDB(\Pi)$ is called *recursion-dependent* if either $p$ is recursive in $\Pi$ or $p$ depends on a recursive predicate of $\Pi$. Let $\mathcal{R}(\Pi)$ denote the set of recursion-dependent predicates in $\Pi$.

The above notions are transferred to grammars in the usual way.

The major notion of this subsection is "recursion-dependent (left-)linear," given below.

*Definition 4.5.* Let $\mathcal{P}^d$ be the limiting mapping defined (for each $\Pi$ and $r$) by (i) $\mathcal{P}^d(\Pi, r) = \mathcal{R}(\Pi)$ if $r$ is a variable renaming of a rule in $\Pi$, and (ii) $\mathcal{P}^d(\Pi, r) = \emptyset$ otherwise. A decomposition $\Pi_1 \cdots \Pi_n$ is called *recursion-dependent (left-)linear* if it is $\mathcal{P}^d$ (left-)linear. A grammar $G$ is called *recursion-dependent (left-)linear* if it is $\mathcal{P}^d_G$ (left-)linear, where $\mathcal{P}^d$ is transferred to grammars in the obvious way.

Roughly speaking, recursion-dependent linear emphasizes the linearity of all recursion-dependent predicates in all rules. Because of the linearity, there is at most one recursive predicate at each step of every derivation for a decomposition of this kind.

The limiting mapping $\mathcal{P}^d$ is more restrictive on rules than $\mathcal{P}^h$ (i.e., $\mathcal{P}^d(\Pi, r) \supseteq \mathcal{P}^h(\Pi, r)$ for all $\Pi$ and $r$). Indeed, let $\Pi$ and $r$ be given. If $r$ is a variable renaming of a rule in $\Pi$ and the head predicate of $r$ occurs in its body, then $\mathcal{R}(\Pi) \supseteq IDB(\{r\})$, and hence $\mathcal{P}^d(\Pi, r) = \mathcal{R}(\Pi) \supseteq IDB(\{r\}) = \mathcal{P}^h(\Pi, r)$. Otherwise, we have $\mathcal{P}^d(\Pi, r) \supseteq \emptyset = \mathcal{P}^h(\Pi, r)$.

Since $\mathcal{P}^d$ is more restrictive than $\mathcal{P}^h$, each recursion-dependent (left-)linear decomposition with 1-rule components is a head (left-)linear decomposition with 1-rule components. We shall see after Lemma 4.1 and Theorem 4.4 that the converses for both the linear and left-linear cases are false.

Theorem 3.1 yields the following analog to Theorem 4.1:

*Theorem 4.3. A program $\Pi$ has a recursion-dependent (left-)linear p-decomposition with 1-rule components iff $L(G_{\Pi,p})$ is generated by a 1-sequential recursion-dependent (left-)linear grammar.*

Turning to our next theorem, we need an auxiliary result and a concept. The auxiliary result is given first.

*Lemma 4.1. The family $\mathcal{L}_1$ of languages generated by (sequential) [left-]linear grammars is identical to the family $\mathcal{L}_2$ of languages generated by (sequential) recursion-dependent [left-]linear grammars.*

PROOF. Clearly, it suffices to show that $\mathcal{L}_2 \subseteq \mathcal{L}_1$. Let $G = (V, \Sigma, P, p_1)$ be a (sequential) recursion-dependent [left-]linear grammar. Since the empty language is in both families, we may assume that $L(G) \neq \emptyset$. For each $p \in V - \mathcal{R}(G)$, let $L_p = \{w \in \Sigma^+ \mid p \Rightarrow_G^* w\}$. By our assumption on grammars (see Section 2.3), $L_p$ is finite and nonempty. We now construct a new grammar from $P$ by replacing each $p \in V - \mathcal{R}(G)$ with every word in $L_p$. Formally, let $G' = (V, \Sigma, P', p_1)$, where

$$P' = \{p \to w_1 \cdots w_k \mid p \to u_1 \cdots u_k \text{ in } P \text{ with each } u_j \in V \cup \Sigma,$$
$$\text{and for each } i, w_i = u_i \text{ if } u_i \in \mathcal{R}(G) \cup \Sigma, \text{ and } w_i \in L_{u_i} \text{ otherwise}\}.$$

Clearly, $L(G') = L(G)$, and all variables occurring in production bodies of $G'$ are recursion-dependent in $G$. Since $G$ is recursion-dependent [left-]linear, each production of $G$ contains at most one recursion-dependent variable in its body. Thus, $G'$ is (sequential) [left-]linear. □

We now give a program $\Pi$ which has a head linear decomposition with 1-rule components, but no recursion-dependent linear decompositions with 1-rule components:

$$\begin{aligned} p(X,Z) &:- p(X,Y_1), c(Y_1,Y_2), p'(Y_2,Z). \\ p(X,Z) &:- p'(X,Y_1), c(Y_1,Y_2), p'(Y_2,Z). \\ p'(X,Z) &:- a(X,Y_1), p'(Y_1,Y_2), b(Y_2,Z). \\ p'(X,Z) &:- a(X,Y), b(Y,Z). \end{aligned}$$

Then $L(G_{\Pi,p}) = \{a^{n_1}b^{n_1}c \cdots ca^{n_m}b^{n_m} \mid m \geq 2, n_i \geq 1\}$ is 1-sequential head linear. By Theorem 4.1, $\Pi$ has a head linear $p$-decomposition with 1-rule components. However, $L(G_{\Pi,p})$ is not linear. [Suppose $L(G_{\Pi,p})$ is linear. Since $a^+b^+ca^+b^+$ is a regular language and linear languages are closed under intersection with regular languages, $L(G_{\Pi,p}) \cap a^+b^+ca^+b^+$ is linear. However, $L(G_{\Pi,p}) \cap a^+b^+ca^+b^+$ is known, e.g.,[12, p. 224], not to be a linear language. Thus, $L(G_{\Pi,p})$ cannot be sequential recursion-dependent linear by Lemma 4.1. Thus, $\Pi$ has no recursion-dependent

linear $p$-decomposition with 1-rule components by Theorem 4.3. (This example also shows that decomposition with 1-rule components is not limited to regular, bounded, and "metalinear" situations.)

The concept is "star-height one," which arises in regular expressions [16].

*Definition 4.6.* A $\{\cup, \bullet, +\}$-*derivation* $L_1, \ldots, L_n$ is said to be *star-height one* if, for each $i$, $L_i = L_j^+$ for some $1 \leq j \leq n$ implies that $L_j$ is finite. A language $L$ is said to be *star-height one* if $L = L_n$ for some star-height one $\{\cup, \bullet, +\}$-derivation $L_1, \ldots, L_n$.

By the corollary of [11, Theorem 1.1], it is easily seen that all regular bounded languages are regular languages of star-height one.

We are now ready for the second theorem of the section.

*Theorem 4.4.* *Let* $\Pi$ *be a program and* $p_1$ *a predicate. Then the following conditions are equivalent:*

(i) $\Pi$ *has a recursion-dependent left-linear* $p_1$*-decomposition with* 1*-rule components.*
(ii) $\Pi$ *has a left-linear* $p_1$*-decomposition with unihead components.*
(iii) $L(G_{\Pi,p_1})$ *is a language of star-height one.*

PROOF. Let $G = G_{\Pi,p_1} = (V, \Sigma, P, p_1)$ and $L = L(G_{\Pi,p_1})$. Note that (i) is equivalent to

(1) $L$ is 1-sequential recursion-dependent left-linear

by Theorem 4.3, and (ii) is equivalent to

(2) $L$ is sequential left-linear

by Theorem 3.1. Hence, it suffices to verify that (1), (2), and (iii) are equivalent.

Using induction on $|V|$, we first prove that (1) implies (iii). Suppose (1) holds. Without loss of generality, we may assume that $G$ is 1-sequential recursion-dependent left-linear. Suppose $p_1$ is recursive in $P$, the case when $p_1$ is not recursive in $P$ being similar and omitted. [A variable $p$ is called *recursive* in a grammar $G_1$ if $p \Rightarrow_{G_1}^* upv$ for $uv \neq \Lambda$.] Since $G$ is 1-sequential, there is a unique production in $P$ such that $p_1$ occurs in both the head and the body. Let $p_1 \to p_1 u$ be this production. Let $p_1 \to v_1, \ldots, p_1 \to v_m$ be the productions in $P$ with $p_1$ as head, where the leftmost symbol of each $v_l$ is not $p_1$.

Clearly, $|V| > 0$. Suppose $|V| = 1$. Since $G$ is 1-sequential recursion-dependent left-linear, $uv_1 \cdots v_m \in \Sigma^*$. Also (see Section 2.3), $v_i \neq \Lambda$ for $1 \leq i \leq m$. Thus, $L(G) = \cup_{i=1}^m \{v_i\} \cup \cup_{i=1}^m v_i u^+$, and is a star-height one language.

Continuing by induction, suppose $k \geq 1$ is an integer such that (iii) holds for $|V| \leq k$. Suppose $|V| = k+1$ and $V = \{p_i \mid 1 \leq i \leq k+1\}$ such that $p_i$ does not depend on $p_j$ for $1 \leq j < i \leq k+1$. Let $j$ $(2 \leq j \leq k+1)$ be fixed, and $G_j = (\{p_i | i \geq j\}, \Sigma, \{p_i \to v \in P \mid i \geq j\}, p_j)$. Since the productions of $G_j$ are contained in $P$, $G_j$ is 1-sequential recursion-dependent left-linear. Clearly, $G_j$ uses at most $k$ variables. By induction, $L(G_j)$ has star-height one. Let $\sigma$ be the language substitution defined by $\sigma(p_i) = L(G_i)$ and $\sigma(a) = \{a\}$ for each $i \geq 2$ and $a \in \Sigma$. Since $G$ is recursion-dependent left linear, $u$ contains no recursion-dependent variable of $G$. Thus, $\sigma(u)$ is finite. Since $G$ is 1-sequential, no $v_i$ contains $p_1$. Hence,

each $\sigma(v_i)$ is the product of star-height one languages, and thus has star-height one itself. Clearly,

$$L = \bigcup_{i=1}^{m} (\sigma(v_i) \cup \sigma(v_i)\sigma(u)^+).$$

Thus, $L$ is star-height one, the induction is extended, and (iii) is verified.

Now, consider that (iii) implies (1). (This part is illustrated in Example 4.2.) Suppose (iii) holds. Then there exists a $\{\cup, \bullet, +\}$-derivation $L_1, \ldots, L_n$ for $L$ of star-height one. Using induction, for $i = 1, \ldots, n$,

(†) we construct a 1-sequential, recursion-dependent left-linear grammar $G_i$ for $L_i$ such that $G_i$ contains no recursive variable if $L_i$ is finite.

Let $G_1 = (\{p_{11}\}, \Sigma, P_1, p_{11})$, where $P_1 = \emptyset$ if $L_1 = \emptyset$ and $P_1 = \{p_{11} \to w \mid w \in L_1\}$ otherwise. Clearly, $G_1$ is a 1-sequential, recursion-dependent left-linear grammar for $L_1$. To facilitate the construction of future grammars, we shall define for each $G_i$ an auxiliary set $A_i$ consisting of a subset of variables of $G_i$. For $G_1$, let $A_1 = \emptyset$ if $L_1 = \emptyset$, and $A_1 = \{p_{11}\}$ otherwise. Clearly, $A_1$ is equal to $\{p' \in V_1 \mid p_{11} \Rightarrow_{G_1}^* p'v \Rightarrow_{G_1} uv$ for some terminal words $u$ and $v\}$. Other $A_i$s will have similar properties.

Continuing by induction, suppose $k \geq 1$ is an integer such that, for each $i \leq k$, (†) holds and the auxiliary set $A_i$ is constructed. Consider $i = k + 1$. Suppose $L_{k+1}$ is constructed from $L_j$ and $L_l$ for some $1 \leq j < l \leq k$. By induction, there exist 1-sequential recursion-dependent left-linear grammars $G_j = (V_j, \Sigma, P_j, p_{j1})$ and $G_l = (V_l, \Sigma, P_l, p_{l1})$ such that $L_j = L(G_j)$ and $L_l = L(G_l)$. Without loss of generality, we may assume that $V_j \cap V_l = \emptyset$. Let $p_{(k+1)1}$ be a new variable and $G_{k+1} = (V_{k+1}, \Sigma, P_{k+1}, p_{(k+1)1})$ be constructed as follows:

($\alpha$) Suppose $L_{k+1} = L_j \cup L_l$. Let $V_{k+1} = \{p_{(k+1)1}\} \cup V_j \cup V_l$, and $P_{k+1} = \{p_{(k+1)1} \to p_{j1}, p_{(k+1)1} \to p_{l1}\} \cup P_j \cup P_l$. Let the auxiliary set $A_{k+1} = A_j \cup A_l$.

($\beta$) Suppose $L_{k+1} = L_j \bullet L_l$. Let $V_{k+1} = \{p_{(k+1)1}\} \cup V_j \cup V_l$, and

$$\begin{aligned} P_{k+1} = {} & \{p_{(k+1)1} \to p_{l1}\} \cup \{p_{lt} \to u \in P_l \mid u \notin \Sigma^+ \text{ or } p_{lt} \notin A_l\} \\ & \cup \{p_{lt} \to p_{j1}u \mid p_{lt} \to u \in P_l, u \in \Sigma^+ \text{ and } p_{lt} \in A_l\} \cup P_j. \end{aligned}$$

Let the auxiliary set $A_{k+1}$ be $A_j$. Informally, the productions generate $L_{k+1}$ as follows:

$p_{(k+1)1}$
$\Rightarrow p_{l1}$ by applying the production in the first set
$\Rightarrow p_{lt}v$ where $v$ is in $\Sigma^*$, by applying productions in the second set
$\Rightarrow^* p_{j1}uv$ by applying a production in the third set
$\Rightarrow^* wuv$ by applying productions in the fourth set.

($\gamma$) Suppose $L_{k+1} = L_j^+$. Since the derivation is of star-height one, $L_j$ must be finite. For each $p$ in $V_j$, let $p'$ be a new variable. Let $h$ be the homomorphism over $(V_j \cup \Sigma)^*$ defined by $h(p) = p'$ for each $p \in V_j$ and $h(b) = b$ for each $b \in \Sigma$. Let $V_{k+1} = \{p_{(k+1)1}\} \cup V_j \cup h(V_j)$, and

$$\begin{aligned} P_{k+1} = {} & \{p_{(k+1)1} \to p_{(k+1)1}p_{j1}, p_{(k+1)1} \to h(p_{j1})\} \cup P_j \\ & \cup \{h(p) \to h(u) \mid p \to u \in P_j\}. \end{aligned}$$

Let the auxiliary set $A_{k+1}$ be $h(A_j)$. By induction, $p_{j1}$ is not recursion-dependent in $G_j$, and hence not recursion-dependent in $G_{k+1}$. Informally, the productions given above produce $L_{k+1}$ as follows:

$p_{(k+1)1}$
$\Rightarrow^* p_{(k+1)1}(p_{j1})^m$ $m \geq 0$, by repeating the first production in the first set
$\Rightarrow p'_{(k+1)1}(p_{j1})^m$ by applying the second production in the first set
$\Rightarrow^* p'_{(k+1)1}u_2 \cdots u_{m+1}$ by applying productions in $P_j$
$\Rightarrow^* u_1 u_2 \cdots u_{m+1}$ by applying productions in $h(P_j)$.

It is easily seen that, for each $i$,

$$A_i = \{p' \in V_i \mid p_{i1} \Rightarrow^*_{G_i} p'v \Rightarrow_{G_i} uv \text{ for some terminal words } u \text{ and } v\}.$$

In all three cases, it is easy to verify that $G_{k+1}$ is a 1-sequential recursion-dependent left-linear grammar for $L_{k+1}$. Thus, the induction is extended, and (1) is proven.

That (1) implies (2) follows from Lemma 4.1.

Finally, consider that (2) implies (1). Thus suppose (2) holds. Without loss of generality, we may assume that $G$ is sequential left-linear. For each recursive variable $p_i \in V$, let $p_{j_i}$ be a new variable. Let $V' = V \cup \{p_{j_i} \mid p_i \text{ is recursive in } G\}$, $P' = \{p \to u \in P \mid p \text{ does not occur in } u\} \cup \{p_i \to p_i p_{j_i}, p_{j_i} \to w \mid p_i \to p_i w \in P\}$, and $G' = (V', \Sigma, P', p_1)$. It is readily verified that $G'$ is a 1-sequential recursion-dependent left-linear grammar which generates $L$. Thus, (1) is proved. □

Clearly, there are regular languages which are not star-height one [13]. By Theorems 4.2 and 4.4, it is easily seen that there are programs which have head left-linear decompositions with 1-rule components, but have no recursion-dependent left-linear decompositions with 1-rule components.

*Example 4.2.* To illustrate "(iii) implies (1)," let $\Pi$ be the following program:

$$\begin{aligned} p(X,Z) &:- p_1(X,Y), p_2(Y,Z). \\ p_1(X,Z) &:- p_1(X,Y), a(Y,Z). \\ p_1(X,Z) &:- a(X,Z). \\ p_2(X,Z) &:- p_2(X,Y), b(Y,Z). \\ p_2(X,Z) &:- b(X,Z). \end{aligned}$$

Then $L(G_{\Pi,p}) = a^+b^+$. One star-height one $\{\cup, \bullet, +\}$-derivation for $L(G_{\Pi,p})$ is: $\{a\}$, $\{b\}$, $\{a\}^+$, $\{b\}^+$, $\{a\}^+\{b\}^+$. The grammars constructed are $G_i$, $1 \leq i \leq 5$. For each $G_i$, the starting symbol is $p_i$ ($p_5 = p$), and the set of productions $P_i$ is given below. Each auxiliary set $A_i$ is also given.

$P_1 = \{p_1 \to a\}$ $A_1 = \{p_1\}$
$P_2 = \{p_2 \to b\}$ $A_2 = \{p_2\}$
$P_3 = \{p_3 \to p_3p_1, p_3 \to p'_1, p_1 \to a, p'_1 \to a\}$ $A_3 = \{p'_1\}$
$P_4 = \{p_4 \to p_4p_2, p_4 \to p'_2, p_2 \to b, p'_2 \to b\}$ $A_4 = \{p'_2\}$
$P_5 = \{p_5 \to p_4, p_4 \to p_4p_2, p_4 \to p'_2, p_2 \to b, p'_2 \to p_3b\} \cup P_3$ $A_5 = \{p'_1\}$

The program decomposition can be constructed from $P_5$ in the standard way.

Suppose $L$ is a context-free language over a one-letter alphabet. Then $L$ is regular and "ultimately periodic" [12]. From this, it readily follows that $L$ is of star-height one. This implies the following:

*Corollary 4.3. Each program with exactly one EDB predicate has a recursion-dependent left-linear p-decomposition with* 1*-rule components (for each predicate p).*

Since it is decidable whether an arbitrary regular language is star-height one [13], we have the following:

*Corollary 4.4. It is decidable if an arbitrary left-linear program has a recursion-dependent left-linear p-decomposition with* 1*-rule components.*

In contrast, we have the following:

*Theorem 4.5. It is undecidable if an arbitrary program with two or more EDB predicates has a recursion-dependent left-linear* $p_1$*-decomposition with* 1*-rule components.*

PROOF. By Theorem 4.4, it suffices to show that it is undecidable if an arbitrary context-free language is star-height one. Let $\Sigma$ be a set of two or more symbols. Since $\Sigma^+$ is star-height one, by Lemma 3.4 it suffices to show that $\{Suf_v(L) \mid v \in \Sigma^+$ and $L$ is star-height one$\}$ is a proper subfamily of the context-free languages. Let $\mathcal{L}$ denote the family of languages of star-height one. Since $\mathcal{L}$ is a subfamily of the regular languages, and thus a proper subfamily of the context-free languages, it suffices to show the following:

(*) For each $L \in \mathcal{L}$ and nonempty word $v$, $Suf_v(L) \in \mathcal{L}$.

By repeated applications, it suffices to consider (*) for $v \in \Sigma$. Let $L$ be in $\mathcal{L}$ and $a$ in $\Sigma$. By induction on the length of $\{\cup, \bullet, +\}$-derivations, it is easily verified that (i) $\mathcal{L}$ is closed under subtraction by singleton sets, and (ii) there exist $L'$ and $L''$ in $\mathcal{L}$ such that $L - \{a\} = aL' \cup L''$ and $L'' \cap a\Sigma^+ = \emptyset$. Clearly, $Suf_a(L) = L'$, and (*) is verified. □

For the recursion-dependent linear case, we have the following result. (The proof is similar to Lemma 4.1, and is thus omitted.)

*Proposition 4.1.* If a program $\Pi$ has a recursion-dependent linear $p$-decomposition, then $L(G_{\Pi,p})$ is linear.

### *4.3. Recursion-One (Left-)Linear Decompositions*

In this subsection, we consider "recursion-one (left-)linear decompositions." These are again special cases of head (left-)linear decompositions.

We start with several concepts.

*Definition 4.7.* Let $\mathcal{P}^o$ be the limiting mapping defined (for each $\Pi$ and $r$) by (i)

$\mathcal{P}^o(\Pi, r) = IDB(\Pi)$ if $r$ is a variable renaming of a rule in $\Pi$ and the head predicate of $r$ occurs in its body, and (ii) $\mathcal{P}^o(\Pi, r) = \emptyset$ otherwise. A decomposition $\Pi_1 \cdots \Pi_n$ is *recursion-one (left-)linear* if it is $\mathcal{P}^o$ (left-)linear. A grammar $G$ is *recursion-one (left-)linear* if it is $\mathcal{P}^o_G$ (left-)linear, where $\mathcal{P}^o$ is transferred to grammars in the obvious way.

Intuitively speaking, recursion-one linear emphasizes that at most one predicate in $IDB(\Pi)$ can appear in the body of a recursive rule. As mentioned in the previous subsection, every derivation tree for each decomposition of this kind has bounded "width" in terms of IDB predicates.

The limiting mapping $\mathcal{P}^o$ is also more restrictive than $\mathcal{P}^h$. [Indeed, let $\Pi$ and $r$ be given. If $r$ is a variable renaming of a rule in $\Pi$ and the head predicate of $r$ occurs in its body, then $\mathcal{P}^o(\Pi, r) = IDB(\Pi) \supseteq IDB(\{r\}) = \mathcal{P}^h(\Pi, r)$. Otherwise, $\mathcal{P}^o(\Pi, r) \supseteq \emptyset = \mathcal{P}^h(\Pi, r)$.] Hence, each recursion-one (left-)linear decomposition with 1-rule components is a head (left-)linear decomposition.

We next recall the concepts of "bounded context-free languages"[7] [10] and "regular bounded languages" [11]. A context-free language $L$ is called *bounded* if there are words $w_1, \ldots, w_n$ such that $L \subseteq w_1^* \cdots w_n^*$. A context-free language is called *regular bounded* if it is regular and bounded.

We now present our characterization of recursion-one (left-)linear decompositions.

*Theorem 4.6. A program $\Pi$ has a recursion-one (left-)linear $p_1$-decomposition with 1-rule components iff $L(G_{\Pi,p_1})$ is (regular) bounded.*

PROOF. We first consider the linear case. By Corollary 3 of [10, Theorem 3.1] and its remark, a language is bounded iff it is generated by a grammar $(\{p_1, \ldots, p_n\}, \Sigma, P, p_1)$, where the variables form a partially ordered set with the following properties:

(a) $p_i \geq p_j$ if and only if $p_i \Rightarrow_G^* up_jv$, where $u$ and $v$ are words over $\Sigma$.
(b) For each $p_i$, there is at most one production where $p_i$ occurs in both the head and the body, *and* each production with $p_i$ as head is either of the form:
   (i) $p_i \rightarrow w$, where $w$ is a terminal word (corresponding to finite sets);
   (ii) $p_i \rightarrow p_j$, where $p_i \geq p_j$ and $p_i \neq p_j$ (corresponding to union);
   (iii) $p_i \rightarrow p_jp_k$, where $p_i \geq p_j$ and $p_i \geq p_k$ and $p_i$, $p_j$, and $p_k$ are mutually distinct (corresponding to concatenation); or
   (iv) $p_i \rightarrow xp_iy$, where $x$ and $y$ are terminal words (corresponding to "nesting").

Clearly, the above grammar is 1-sequential recursion-one linear. Thus, a language is bounded iff it is 1-sequential recursion-one linear. By Theorem 3.1, it follows that a program $\Pi$ has a recursion-one linear $p_1$-decomposition with 1-rule components iff $L(G_{\Pi,p_1})$ is bounded.

To consider the left-linear case, let $L$ be a context-free language. By the corollary of [11, Theorem 1.1], it is easily seen that $L$ is regular bounded iff $L$ is generated by a context-free grammar of the form given above, except that (iv) is replaced by

[7] The boundedness of context-free languages should not be confused with the "boundedness" of Datalog programs.

"$p_i \rightarrow p_i y$, where $y$ is a terminal word." The remainder of the proof can be given as for the linear case by making the obvious changes. □

To illustrate the linear case, consider the following program $\Pi$:

$$r_1 : p(X, Z) \text{ :- } a(X, Y_1), a(Y_1, Y_2), p(Y_2, Y_3), b(Y_3, Y_4), b(Y_4, Y_5), b(Y_5, Z).$$
$$r_2 : p(X, Z) \text{ :- } a(X, Y_1), a(Y_1, Y_2), a(Y_2, Y_3), p(Y_3, Y_4), b(Y_4, Y_5), b(Y_5, Z).$$
$$r_3 : p(X, Z) \text{ :- } c(X, Z).$$

Then $L(G_{\Pi,p}) \subseteq a^*cb^*$, and is thus bounded. By Theorem 4.6, $\Pi$ has a recursion-one linear $p$-decomposition with 1-rule components. In fact, $\{r_1\}\{r_2\}\{r_3\}$ is one such decomposition. Note that $\{r_1, r_2\}$ is an SCC and is not a recursion-one linear size prime.

By [10, Theorem 5.2], it is decidable if an arbitrary regular or context-free language is bounded. Thus, we have:

*Corollary 4.5. It is decidable if an arbitrary (left-linear) program has a recursion-one (left-)linear p-decomposition with 1-rule components.*

We conclude this section by discussing some relationships (in addition to those given in the previous subsection) among the three kinds of decompositions. As mentioned at the beginning of this subsection, each recursion-one (left-)linear decomposition with 1-rule components is also a head (left-)linear decomposition. However, the converse is false. Indeed, let $\Pi$ be a program such that $L(G_{\Pi,p}) = \{a, b\}^+$. Since $L(G_{\Pi,p})$ is regular, it follows from Theorem 4.2 that $\Pi$ has a head left-linear $p$-decomposition with 1-rule components. Since $L(G_{\Pi,p})$ is not bounded [12], $\Pi$ does not have a recursion-one (left-)linear $p$-decomposition with 1-rule components (by Theorem 4.6).

Similar to the argument given after Example 4.2, it is easily verified that each regular bounded language has star-height one. By Theorems 4.4 and 4.6, it follows that each program having a recursion-one left-linear $p$-decomposition with 1-rule components has a recursion-dependent left-linear $p$-decomposition with 1-rule components.

Recursion-dependent (left-)linear decompositions with 1-rule components are not comparable with recursion-one linear decompositions with 1-rule components. Indeed, let $\Pi$ be as above. As already noted, $\Pi$ has no recursion-one linear $p$-decomposition with 1-rule components. Since $L(G_{\Pi,p})$ has star-height one, $\Pi$ has a recursion-dependent left-linear $p$-decomposition with 1-rule components by Theorem 4.4. On the other hand, let $\Pi'$ be a program such that $L(G_{\Pi',p}) = \{a^n b^n a^m b^m \mid n, m \geq 1\}$. By Theorem 4.6 and Proposition 4.1, it follows that $\Pi'$ has a recursion-one linear $p$-decomposition with 1-rule components, but no recursion-dependent (left-)linear $p$-decomposition with 1-rule components.

## 5. CONCLUDING REMARKS

This paper introduced the notion of $\mathcal{P}$ (left-)linear decompositions of chain Datalog programs, and studied such decompositions with 1-rule components. The results developed were mainly on three themes: characterizations, decidability, and pri-

mality. The basic theorems obtained related decompositions with corresponding properties (such as 1-sequential, bounded, regular, and star-height one) of the associated context-free languages.

One direction for further research into decomposition is complexity. Issues of interest include (i) the complexity of decompositions (such as the number of rules, which were briefly discussed here), and (ii) the efficiency of evaluating decompositions (as compared with the efficiency of evaluating nondecomposed programs). Another topic for study is the uniqueness of "factorization." For example, what does uniqueness mean? And are there "reasonable conditions" which guarantee it?

## APPENDIX

We say a CFG $G = (V, \Sigma, P, p_1)$ has *k-SCC components*, where $k$ is a positive integer, if each SCC of $P$ has at most $k$ productions. Clearly, if $L$ is a $k$-sequential CFL, then $L$ is generated by a CFG having $k$-SCC components.

Let $m \geq 2$ and $\Sigma = \{a_1, \ldots, a_m, c\}$, where $a_1, \ldots, a_m$, and $c$ are pairwise distinct symbols. Let $u^R$ denote $a_{i_n} a_{i_{n-1}} \cdots a_{i_1}$ if $u = a_{i_1} \cdots a_{i_{n-1}} a_{i_n}$, $1 \leq i_1, \ldots, i_n \leq m$. Let $L_0 = \{ucu^R \mid u \in (\Sigma - \{c\})^+\}$. Recall that $Suf_v(L) = \{u \neq \Lambda \mid vu \in L\}$. We now show:

*Proposition* A. For each word $v$ over $\Sigma$ and each CFL $L$ generated by a CFG having $(m-1)$-SCC components (or which is $(m-1)$-sequential), $Suf_v(L) \neq L_0$. Consequently, $L_0$ cannot be generated by a CFG which either has $(m-1)$-SCC components or is $(m-1)$-sequential (by letting $v = \Lambda$).

PROOF. Assume to the contrary that there exist a word $v$ over $\Sigma$ and a CFL $L$ generated by a CFG $G = (V, \Sigma, P, p_1)$ having $(m-1)$-SCC components such that $Suf_v(L) = L_0$. Let $V_0 = \{p \in V \mid$ there is a leftmost derivation $p_1 \Rightarrow_G^* vx$ such that $p$ occurs in $x\}$, and let $P_0 = \{p \to u \in P \mid p \in V_0\}$. In other words, $P_0$ consists of all productions of $P$ that can be applied in a leftmost derivation after the prefix $v$ is generated. Let $s_0$ be the maximum of the body lengths of the productions in $P$. We shall establish a sequence of claims, the first being:

(1) By transforming $G$ if necessary, each production in $P_0$ may be assumed to contain at most one variable, and only one occurrence of that, in its body.

Since each variable that can generate exactly one terminal word may be replaced by the terminal word it generates, it suffices to show the following: If $p_1 \Rightarrow_G^* vx$, then $x$ contains at most one variable that can generate two or more terminal words. Indeed, assume the contrary, i.e., there exist $w_1', w_2', w_1'', w_2''$ in $\Sigma^*$ and a derivation $p_1 \Rightarrow_G^* vu_1 p' u_2 p'' u_3$ such that $w_1' \neq w_2'$, $w_1'' \neq w_2''$, $p' \Rightarrow_G^* w_j'$ and $p'' \Rightarrow_G^* w_j''$ for $j = 1, 2$. By our assumption on grammars (see Section 2.3), there exist $u_i'$ in $\Sigma^*$ such that $u_i \Rightarrow_G^* u_i'$ for $1 \leq i \leq 3$. Therefore, $p_1 \Rightarrow_G^* vu_1' w_i' u_2' w_j'' u_3'$ for $1 \leq i, j \leq 2$. But some of the four words $u_1' w_i' u_2' w_j'' u_3'$ $(1 \leq i, j \leq 2)$ are not palindromes, a contradiction.

The second claim is the following.

(2) Let $s_1 = (2|v| + 1)|V|s_0$. Then $|x| \leq s_1$ for each leftmost derivation $p_1 \Rightarrow_G w_1 \Rightarrow_G \cdots \Rightarrow_G w_k$, where $w_k$ is the first among $w_1, \ldots, w_k$ to have $v$ as a prefix and $w_k = vx$.

We first show the following: If a loop in the leftmost derivation above contributes to $x$, then it must also contribute to $v$. More precisely, if

$$p_1 \overset{*}{\underset{G}{\Rightarrow}} v_1 p u_1 \overset{*}{\underset{G}{\Rightarrow}} v_1 v_2 p u_2 u_1 \overset{*}{\underset{G}{\Rightarrow}} v_1 v_2 u_3 u_2 u_1$$

is an initial segment of the above leftmost derivation, where $v_1 v_2$ is over $\Sigma$ with $|v_1 v_2| < |v|$, $|v_1 v_2 u_3| \geq |v|$, and $u_2 \neq \Lambda$, then $v_2 \neq \Lambda$. Indeed, suppose otherwise. There exist $u_i'$ over $\Sigma$ such that $u_i \Rightarrow_G^* u_i'$ for $1 \leq i \leq 3$. Then $p_1 \Rightarrow_G^* v_1 u_3' (u_2')^j u_1'$ for each positive integer $j$. But not all the words $v_1 u_3' (u_2')^j u_1'$ have the form $vwcw^R$. In fact, clearly, $c$ must occur in $u_3' u_2' u_1'$. If $c$ occurs in $u_3' u_1'$, then some of these words have more letters on one side of $c$ than the other side. If $c$ occurs in $u_2'$, then there is more than one $c$ after $v$ whenever $j > 1$.

Returning to (2), note that each $w_i$ contains at most one variable $p$ which contributes to $x$ in the derivation, i.e., $p$ leads to the inclusion in $x$ of at least one symbol in $\Sigma \cup V$. Since the grammar is $\Lambda$-free, all variables in the derivation must eventually (perhaps outside of the above derivation) lead to a word in $\Sigma^+$. Since each loop which contributes to $x$ must also contribute to $v$, the derivation can be divided into at most $|v|$ loops which contribute to $x$ and do not properly contain loops which contribute to $x$, plus up to $|v|+1$ segments not containing loops which contribute to $x$. Since each loop and segment generates at most $|V|s_0$ symbols, (2) follows.

Let $P_1, \ldots, P_n$ be the SCCs of $P$. Then $|P_i| \leq m-1$ for each $i$ since $G$ has $(m-1)$-SCC components. By reordering if necessary, we may assume that if $i < j$, $r_i \in P_i$, and $r_j \in P_j$, then the head of $r_i$ does not occur in the body of $r_j$. Let $s = |V_0|s_0 + s_1$. By diagonalization, we now construct a word $vwcw^R \in L$ which the grammar $G$ cannot generate. Intuitively, if there is a derivation for $vwcw^R$, then no production in $P_i$ can be used in generating the $[3(i-1)s+1, 3is]$ interval of $w$. Thus, only productions from $\cup_{j=i+1}^{n} P_j$ can be used there. Consequently, the productions in $P_i$ cannot be applied afterwards. Formally, for each $i$, $1 \leq i \leq n$, let $Prefix_1(P_i) = \{a_j \in \Sigma \mid p \rightarrow a_j u \text{ is in } P_i\}$. Since $|P_i| \leq m-1$, it is clear that $\{a_1, \ldots, a_m\} - Prefix_1(P_i) \neq \emptyset$. For each $i$ $(1 \leq i \leq m)$, let $b_i$ be a fixed element of $\{a_1, \ldots, a_m\} - Prefix_1(P_i)$. Let $w$ be the word defined by $w(j) = b_i$ for $1 \leq j \leq 3(n+1)s$, where $i$ satisfies $3(i-1)s+1 \leq j \leq 3is$. (We use $w(j)$ to denote the $j$th letter in $w$.)

Assume there exists a leftmost derivation for $vwcw^R$. Consider the segment of this derivation starting from where $v$ is *first* completely generated, i.e.,

(3) $vx_0 \Rightarrow_G vx_1 \Rightarrow_G \cdots \Rightarrow_G vx_t = vwcw^R$.

By (1), for each $k$ $(0 \leq k < t)$, $x_k$ may be written as $y_k p_{\beta_k} z_k$ where $y_k$ and $z_k$ are terminal words and $p_{\beta_k}$ is a variable. Call a pair $(i,j)$ a loop if $\beta_i = \beta_j$ and $i < j$. By removing each loop in the derivation where all the participating words have equal lengths if necessary, we may assume that $|x_i| < |x_j|$ whenever $(i,j)$ is a loop. Clearly, the following condition must hold:

(4) If $i_0 < j_0$ and there is no loop $(i,j)$ with $i_0 \leq i < j \leq j_0$, then $|x_{j_0}| \leq |x_{i_0}| + s$.

Let $t_0 = \max\{j \mid (i,j) \text{ is a loop in (3)}\}$. Clearly, $t_0$ exists. [Otherwise, $|x_0| \leq s$ by (2). Then $|x_t| \leq |x_0| + s \leq 2s < 6(n+1)s + 1 = |wcw^R|$. This contradicts the fact that $x_t = wcw^R$.] Furthermore, $t_0 < t$. We now establish the following claim:

(5) $c$ does not occur in $x_{t_0}$.

By our assumption on grammars (see Section 2.3), there exists $w_0 \in \Sigma^+$ such that $p_{\beta_{t_0}} \Rightarrow_G^* w_0$. By the definition of $t_0$, there exists an integer $t_1$ such that $(t_1, t_0)$ is a loop. Since $|x_{t_1}| < |x_{t_0}|$, there exists $u_1u_2 \in \Sigma^+$ such that $y_{t_0}p_{\beta_{t_0}}z_{t_0} = y_{t_1}u_1p_{\beta_{t_0}}u_2z_{t_1}$ and $\beta_{t_0} = \beta_{t_1}$. If $c$ occurs in $x_{t_0}$, then at most one of $y_{t_0}w_0z_{t_0}$ and $y_{t_0}u_1w_0u_2z_{t_0}$ can be a palindrome, a contradiction. Thus, (5) is proven.

Let $i_0 \leq i_1 \leq \cdots \leq i_{t_0-1}$ be integers such that a production $p_{\beta_k} \rightarrow u_kp_{\beta_{k+1}}u'_k$ from $P_{i_k}$ is applied in obtaining $y_kp_{\beta_k}z_k \Rightarrow_G y_{k+1}p_{\beta_{k+1}}z_{k+1}$. Then $y_{k+1} = y_ku_k$. Using induction, we now show that

(6) $|y_k| \leq 3(i_k - 1)s + 2s$ for $0 \leq k \leq t_0$.

For $k = 0$, $|x_0| \leq s$ by (2), so $|y_0| \leq |x_0| \leq s \leq 3(i_k - 1)s + 2s$. Suppose $k_1$ is an integer $(0 \leq k_1 < t_0)$ such that (6) holds for $k = k_1$. Consider $k = k_1 + 1$. Two cases arise:

(i) $|y_{k_1}| \leq 3(i_{k_1} - 1)s + s$. Since $|u_{k_1}| \leq s_0 \leq s$ and $y_k = y_{k_1}u_{k_1}$, it follows that $|y_k| \leq |y_{k_1}| + s \leq 3(i_{k_1} - 1)s + 2s \leq 3(i_k - 1)s + 2s$.

(ii) $|y_{k_1}| > 3(i_{k_1} - 1)s + s$. By induction, $|y_{k_1}| \leq 3(i_{k_1} - 1)s + 2s$. Thus, $3(i_{k_1} - 1)s + s < |y_{k_1}| \leq 3(i_{k_1} - 1)s + 2s$. Two subcases arise: (a) $u_{k_1} = \Lambda$. Now, $i_{k_1} \leq i_k$. Hence, $|y_k| = |y_{k_1}| \leq 3(i_{k_1} - 1)s + 2s \leq 3(i_k - 1)s + 2s$. (b) $u_{k_1} \neq \Lambda$. Since $w(j) \notin Prefix_1(P_{i_{k_1}})$ for $3(i_{k_1} - 1)s + s < j \leq 3(i_{k_1} - 1)s + 2s$, we have $i_k > i_{k_1}$. Thus, $|y_k| \leq |y_{k_1}| + s \leq 3(i_{k_1} - 1)s + 2s + s = 3i_{k_1}s \leq 3(i_k - 1)s$. In all cases, the induction is extended, and (6) is verified.

Finally, $|y_{t_0}| \leq 3(i_{t_0} - 1)s + 2s$ by (6). There exists $w_t$ such that $x_t = y_{t_0}w_tz_{t_0}$. By (5), $c$ occurs in $w_t$. By (4), $|w_t| \leq s$. Let $w'$ and $w''$ be words such that $x_t = w'cw''$. Then $|w'| \leq |y_{t_0}| + |w_t| \leq 3(i_{t_0} - 1)s + 2s + s \leq 3ns < 3(n+1)s = |w|$, contradicting the assumption that $x_t = wcw^R$. □

The authors wish to thank Q. Li, J. M. Robson, and J. Su for providing useful comments on earlier drafts of the paper, and the anonymous referees for careful reading and helpful comments on an earlier draft of the paper.

## REFERENCES

1. Afrati, F. and Cosmadakis, S. S., Expressiveness of restricted recursive queries, in *Proc. ACM SIGACT Symp. on the Theory of Computing*, 1989, pp. 113–126.
2. Afrati, F. and Papadimitriou, C. H., The parallel complexity of simple chain queries, in: *Proc. ACM Symposium on Principles of Database Systems*, 1987, pp. 210–213.
3. Beeri, C., Kanellakis, P., Bancilhon, F., and Ramakrishnan, R., Bounds on the propagation of selection into logic programs, in: *Proc. ACM Symposium on Principles of Database Systems*, 1987, pp. 214–226.
4. Dong, G., On the composition and decomposition of datalog program mappings, in: *Proc. of 1988 ICDT;* also as *Lecture Notes in Computer Science No. 326* (M. Gyssens, J. Paredaens, and D. Van Gucht, eds.), 1988, pp. 87–101.
5. Dong, G., On distributed processibility of datalog queries by decomposing databases, in: *Proc. ACM SIGMOD International Conference on Management of Data,* Oregon, 1989, pp. 26–35.

6. Dong, G., On datalog linearization of chain queries, in: J. D. Ullman, ed., *Theoretical Studies in Computer Science*, Academic Press, 1991, pp. 181–206.
7. Dong, G. and Ginsburg, S., On the decomposition of datalog program mappings, *Theoretical Computer Science* 76(1):143–177, (Oct. 1990).
8. Ginsburg, S. and Rice, H. G., Two families of languages related to ALGOL, *Journal of ACM* 9:350–371, (1962).
9. Ginsburg, S. and Rose, G. F., Some recursively unsolvable problems in ALGOL-like languages, *JACM* 10:29–47, (1963).
10. Ginsburg, S. and Spanier, E. H., Bounded ALGOL-like languages, *Trans. Amer. Math. Soc.*, 113:333–368, (1964).
11. Ginsburg, S. and Spanier, E. H., Bounded regular sets, *Proc. Amer. Math. Soc.* 17:1043–1049, (1966).
12. Harrison, M. A. *Introduction to Formal Language Theory*, Addison-Wesley, Reading, MA, 1978.
13. Hashiguchi, K. Regular languages of star height one, *Information and Control* 53:199–210, (1982).
14. Hunt, H. B. III and Rosenkratz, D. J., Computational parallels between the regular and context-free languages, *SIAM J. Computing* 7(1):99–114, (1978).
15. Ioannidis, Y. E. and Wong, E., Transforming nonlinear recursion to linear recursion, in: L. Kerschberg, ed., *Proc. of Second International Conference on Expert Database Systems*, 1988, pp. 187–207.
16. Lewis, H. R. and Papadimitriou, C. H., *Elements of the Theory of Computation*, Prentice-Hall, 1981.
17. Lloyd, J. W., *Foundations of Logic Programming, Second Edition*, Springer-Verlag, 1987.
18. Naughton, J. F., One-sided recursions, in: *Proc. ACM Symposium on Principles of Database Systems*, 1987, pp. 340–348.
19. Naughton, J. F., Compiling separable recursions, in: *Proc. ACM SIGMOD International Conference on Management of Data*, 1988, pp. 312–319.
20. Naughton, J. F., Ramakrishnan, R., Sagiv, Y., and Ullman, J. D., Argument reduction by factoring, in: *Proc. International Conference on Very Large Databases*, 1989, pp. 173–182.
21. Plambeck, T., Semigroup techniques in recursive query optimization, in: *Proc. ACM Symposium on Principles of Database Systems*, 1990, pp. 145–153.
22. Ramakrishnan, R., Sagiv, Y., Ullman, J. D., and Vardi, M. Y., Proof-tree transformation theorems and their applications, in: *Proc. ACM Symposium on Principles of Database Systems*, 1989, pp. 172–181.
23. Saraiya, Y. P., Linearizing nonlinear recursions in polynomial time, in: *Proc. ACM Symposium on Principles of Database Systems*, 1989, pp. 182–189.
24. Saraiya, Y. P., Hard problems for simple logic programs, in: *Proc. ACM SIGMOD International Conf. on Management of Data*, 1990, pp. 64–73.
25. Saraiya, Y. P., Polynomial-time program transformations in deductive databases, in: *Proc. ACM Symposium on Principles of Database Systems*, 1990, pp. 132–144.
26. Shmueli, O., Decidability and expressiveness aspects of logic queries, in: *Proc. ACM Symposium on Principles of Database Systems*, 1987, pp. 237–249.
27. Ullman, J. D., *Principles of Database and Knowledge-Base Systems*, Vol. I, Computer Science Press, 1988.

28. Ullman, J. D., *Principles of Database and Knowledge-Base Systems*, Vol. II, Computer Science Press, 1989.
29. Ullman, J. D., and Van Gelder A., Parallel complexity of logical query programs, *Algorithmica* 3:5–42, (1988).
30. Zhang, W., Yu, C. T., and Troy, D., Necessary and sufficient conditions to linearize doubly recursive programs in logic databases, *ACM Transactions on Database Systems* 15(3):459–482, (1990).